PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06K

A2

(11) International Publication Number: **WO 98/50874**

(43) International Publication Date: 12 November 1998 (12.11.98)

(21) International Application Number: PCT/US98/09018

(22) International Filing Date: 7 May 1998 (07.05.98)

(30) Priority Data:
08/854,262 9 May 1997 (09.05.97) US

(71) Applicant: KEYLABS, INC. [US/US]; 633 South 550 East, Provo, UT 84606 (US).

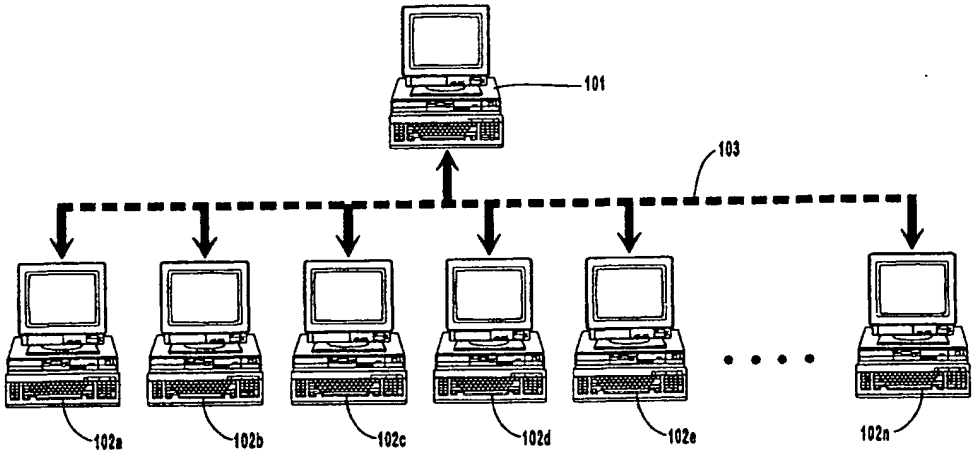
(72) Inventors: TURPIN, Kevin, J.; 1154 N. 660 W., Orem, UT 84057 (US). CLARK, Christopher, P.; 969 Pasque Drive, Salt Lake City, UT 84123 (US).

(74) Agents: SADLER, Lloyd, W. et al.; McCarthy & Sadler, LC, Suite 100, 39 Exchange Place, Salt Lake City, UT 84111 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, GW, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: METHOD AND SYSTEM FOR CLIENT/SERVER AND PEER-TO-PEER DISK IMAGING



(57) Abstract

A method and system for imaging data between two or more digital computers across a computer network is described, where the digital computers transfer data in a peer-to-peer mode and/or a client/server mode upon command of the operator. This invention addresses the problem of managing, updating and installing executable software, such as operating systems, utilities and application software packages on a large number of networked computer systems. By using this invention properly, a system operator can transfer data stored on a single computer system to all or some of the computer system connected to the first system over a computer network and can do so without expensive electronic server equipment. Moreover, this invention provides the capability of transferring data as files, sectors or cylinders of disk media, thereby permitting a single operator to, through a generally automated procedure, simultaneously install new system software, configuration files and executive files on many computers. This invention provides an important improvement in the operation, maintenance and control of large computer networks, although it applies and works equally well in small network applications. In its best mode of operation this invention is performed on standard digital computer systems through the use of special purpose computer software.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

METHOD AND SYSTEM FOR CLIENT/SERVER AND PEER-TO-PEER DISK IMAGING

Background of the Invention

Field of the Invention.

This invention relates to the systems and methods for copying or mirroring the binary data on one computer hard disk drive over a computer network to one or many other computer hard disk drives. More specifically, this invention provides a process or method for installing and/or distributing software from one computer system to one or more other computer systems over a computer network. Furthermore, this invention provides a system for solving the often tedious problems of installing computer software and distributing computer system files to a number of computer systems, providing a mechanism for fast software distribution on networked computers by eliminating the need to use the installation utilities of each application program to install the software package individually on each computer.

Description of Related Art.

It is commonly known in the related art to transfer computer files from one computer hard disk to another computer hard disk. Similarly, it is well known to transfer files from computer to computer over a computer network. Likewise, computer network vendors have created tool sets, for use in their own labs, to transfer data from a master computer disk drive to an image file on a file server and then to download the data to the target or slave computers. Other existing tools will use the file server to broadcast data from an image file to the target or slave computers simultaneously (in parallel). Examples of these tools have been disclosed at conferences and with customers by

Novell, Inc.

This invention has substantial and important advantages over prior known approaches. This invention not only can use a client/server model, it can also use a peer-to-peer model not available with computer network vendor tool sets or prior used disk-to-disk copying. Unlike, prior approaches, this invention can be used without a network file server and still can copy computer data from one computer hard disk to many computer hard disks over a computer network, by using the peer-to-peer mode of operation. Moreover, the peer-to-peer mode of operation provides important advantages in terms of transfer speed and lower cost. The speed advantage is realized by using a one step process rather than the two step process required for the client/server model of operation. In the invention's peer-to-peer mode, the data is distributed from the master computer to the slave computers in a single step. The cost advantage is achieved by not requiring a network file server to accomplish the one to many data imaging. Network file servers are costly to purchase, install and maintain.

For general background art the reader is directed to United States Patent Nos. 4,866,664, 4,872,006, 5,249,290, 5,325,527, 5,396,613, 5,421,009, 5,438,671, 5,452,459, 5,459,837, 5,461,721, 5,465,351, 5,491,694, 5,495,611, 5,506,902, 5,513,126, 5,513,314, 5,515,508, 5,515,510, 5,517,645, 5,517,668, 5,522,041, 5,526,490, 5,528,757, 5,537,533, 5,537,585, 5,542,046 each of which is hereby incorporated by reference in its entirety for the material disclosed therein.

U.S. Patent No. 4,866,664 discloses an interprocessor message communication synchronization apparatus and method for a plurality of processors connected to a system bus where one processor desiring to send a control signal to another processor, broadcasts

an input/output write instruction on the system bus along with the address of the receiving processor and a data field representative of the control signal to be transmitted.

U.S. Patent No. 4,872,006 discloses a data transmission system in which data are transmitted among plural stations.

U.S. Patent No. 5,249,290 discloses a method of and apparatus for operating a client/server computer network to access shared server resources in response to service requests from client computers connected to the network.

U.S. Patent No. 5,325,527 discloses a client/server communication system utilizing a self-generating nodal network wherein the method includes the steps of creating a server nodal network tree which includes the steps of generating a server root node which includes both process steps for communicating to an operating system and service nodes, and process steps for building service nodes which correspond to servers within the client/server system, each service node include both process steps for advertising a service to the server root node and process steps for building a topic node which includes both process steps for accessing a server and process steps for building a job node for storing a job request.

U.S. Patent No. 5,396,613 discloses a method for error recovery in client/server distributed processing systems using cascaded servers.

U.S. Patent No. 5,421,009 discloses a method for remote installation of software over a computer network, allowing the user to interactively select each remote computer system for software installation, or to provide a file containing a list of all remote computer systems.

U.S. Patent No. 5,438,671 discloses a two-computer system and method where

data is transferred between the computers as complete disk images rather than as files.

U.S. Patent No. 5,452,459 discloses a method and apparatus for allocating server access in a distributed computing environment using a scheduling process.

U.S. Patent No. 5,459,837 discloses a method and system for monitoring the performance of servers across a network and for suggesting an appropriate server to a client requesting a service, wherein a plurality of probes are placed in various clients in the network by a Broker-Performance Mechanism.

U.S. Patent No. 5,461,721 discloses a system for transferring data between input/output devices and main or expanded storage under dynamic control of independent indirect address words.

U.S. Patent No. 5,465,351 discloses a method and system for memory management of a client/server computing network.

U.S. Patent No. 5,491,694 discloses an apparatus and method for establishing "virtual connections" through a packet switched data communications network, the network including a plurality of end systems and switches connected by links, to allocate a shared resource among competing devices.

U.S. Patent No. 5,495,611 discloses a method and apparatus for dynamically loading an ABIOS device support layer in a computer system.

U.S. Patent No. 5,506,902 discloses a data broadcasting system for the low-cost delivery of character-heavy data such as newspapers and magazines.

U.S. Patent No. 5,513,126 discloses a method for a sender to automatically distribute information to a receiver on a network using devices and communication channels defined in the receiver profiles.

U.S. Patent No. 5,513,314 discloses a fault tolerant NFS server system and mirroring protocol for the retrieval of data files including a client system connected to a data communication network.

U.S. Patent No. 5,515,508 discloses a object-oriented client/server facility (CSF) and networking service facility (NSF) interfacing between application programs residing in client and server nodes of a distributed services network.

U.S. Patent No. 5,515,510 discloses a communications internetwork system connecting a client node array to a resource array.

U.S. Patent No. 5,517,645 discloses a method and system for managing the connection of client components to an interface implemented by a sever component.

U.S. Patent No. 5,517,668 discloses a distributed computing system having a distributed protocol stack.

U.S. Patent No. 5,522,041 discloses a data processor and a data transfer method for efficiently transferring data between a plurality of information processing devices as in a client server system.

U.S. Patent No. 5,526,490 discloses a data transfer control unit using a control circuit to achieve high speed data transfer.

U.S. Patent No. 5,528,757 discloses a communication network system in which a plurality of information processing equipments are connected with a communication line for communication of a message.

U.S. Patent No. 5,537,533 discloses a system for remote mirroring of digital data from a primary network server to a remote network server, which includes a primary data transfer unit and a remote data transfer unit which are connected one with another by a

conventional communication link.

U.S. Patent No. 5,537,585 discloses a data storage management system for networked interconnected processors, including a local area network and a storage server.

U.S. Patent No. 5,542,046 discloses a peer to peer connection authorizer which includes a system authorizer mechanism, a client connection manager, and a server connection manager.

None of these prior related art references discloses a system for imaging (or mirroring) binary data from one computer hard disk drive over a computer network to one or many other computer hard disk drives either through a client/server mode or a peer-to-peer mode, to provide a mechanism or process for rapid software distribution on networked computers which eliminates the necessity of using each application's install utility individually on each computer.

Summary of the Invention

It is desirable to provide a method and system for installing computer software and computer data files on more than one computer simultaneously, over a network, where the method can function upon user command either under a client/server model or a peer-to-peer model of operation.

Accordingly, it is the primary object of this invention to provide a process for installing computer application programs on multiple computer systems simultaneously over a computer network.

It is a further object of this invention to provide a process for mirroring data files on multiple computer systems simultaneously over a computer network.

It is further object of this invention to provide a method for copying binary

computer data from one computer hard disk drive to one or more other computer hard disk drives, which does not require the use of a computer file server.

It is a further object of this invention to provide a system for mirroring computer data from one computer disk drive to another computer disk drive in a peer-to-peer computer communications model.

It is a further object of this invention to provide a system for mirroring computer data from one computer disk drive to another computer disk drive in a client/server computer communications model.

It is a further object of this invention to provide a method for mirroring computer data between one computer disk drive to another computer disk drive that permits the user to select between a client/server and peer-to-peer computer communications.

It is a further object of this invention to provide a method and system for mirroring computer data from disk drive to disk drive that improves the cost performance of client/server systems available in the art.

It is a further object of this invention to provide a computer data mirroring system that incorporates data compression.

It is a further object of this invention to provide a computer data imaging system that has the capability of imaging a single disk partition, multiple disk partitions, or the entire hard disk, from one computer hard disk to one or more other computer hard disks simultaneously.

Additional objects, features and advantages of this invention will become apparent to persons of ordinary skill in the art upon reading the remainder of the specification and upon referring to the attached figures.

These objects are achieved by a set of two computer programs, referred to by the inventor as IMGBLSTR and IMGSLAVE. The IMGBLSTR program is the primary control program while the IMGSLAVE program is used for listening for data from the IMGBLSTR program across the network and writing such data to the local disk drive.

The IMGBLSTR program operates in five modes of operation. These are:

1. IMGBLSTR reads the data from the disk drive of the master computer, compresses the data, and writes it (uploads it) to an "image file" resident on a network file server. This is the client/server model or mode of operation.
2. In connection with performing mode 1 above, IMGBLSTR broadcasts (simultaneously sends in parallel) the data that is being uploaded to the image file on the network to all computers running the IMGSLAVE program. This mode of operation is a combination of client/server and peer-to-peer.
3. IMGBLSTR reads the data from the local computer disk drive and broadcasts it on the wire (network) to computers running the IMGSLAVE program. The IMGBLSTR program does not upload the data to an image file on a network file server, rather the data goes directly to the slave computers running the IMGSLAVE program. This is the peer-to-peer mode of operation.
4. IMGBLSTR reads the data from an image file located on a network file server, decompresses it, and writes it to its own local computer hard disk drive. This the download process for the client/server model.
5. While performing operation mode 4 above, IMGBLSTR broadcasts the data that is being downloaded from the image file on the network to all computers running the IMGSLAVE program. This is a combination client/server and peer-to-peer mode.

The IMGBLSTR program determines which mode to use through either a series of menu options presented to the operator, or through commands from the operator entered on the command line when the program is launched.

The IMGSLAVE program operates in only one mode of operation. Specifically, it opens a communication socket on the network, listens for data received on that socket, and then processes the data received on the socket. Each packet of data received on the socket contains a command field which tells IMGSLAVE what the data contained in the packet is used for and how the data is to be processed. The commands in the command field are:

Command	Performed by	Function
Drive Geometry	Master	Compare geometry of master image with slave
RSVP	Slave	Response to master to indicate participation in download
Conform Download	Master	Acknowledgment that slave has joined the process and that master knows slave is ready
Sector Data	Master	Write data to receive buffer
Sector Data & Flush	Master	Write data to receive buffer and flush data to disk
Skip Track	Master	No data in current track - Skip this track
Resend Request	Slave	Slave missed data - Please resend

End of Data	Master	Master is finished sending data - Ready for a slave request
Done	Master	Image complete - Exit program
Disconnect	Slave	Slave response to master "done" command - Slave disconnecting
Disconnect Acknowledge	Master	Master acknowledges slave disconnect

Current source code for both the IMGBLSTR and the IMGSLAVE programs are included and listed in the source code section of the detailed description of the invention.

Brief Description of the Drawings

Figure 1 depicts a system diagram of a computer network having a source (master) computer and a number of destination (slave) computers connected to each other electronically.

Figure 2 depicts a system diagram of a computer network using the prior approaches utilizing client/server modes of operation for transferring data from one computer to other computers across a network.

Figure 3 depicts the top level state diagram of the master computer ("IMGBLSTR") program / process component of the invention.

Figure 4 depicts the top level state diagram of the slave computer ("IMGSLAVE") program / process component of the invention.

Figure 5 depicts the top level process flow chart of the master computer component of the invention.

Figure 6 depicts the detailed flow diagram of the process menus steps of the master

computer component of the invention.

Figure 7 depicts the detailed flow diagram of the upload image step of the master computer component of the invention.

Figure 8 depicts further detail of the read head step of the upload image section of the master computer component of the invention.

Figure 9 depicts further detail of the broadcast head step of the upload image section of the master computer component of the invention.

Figure 10 depicts further detail of the download image step of the master computer component of the invention.

Figure 11 depicts further detail of the fill compress buffer from file step of the down load image step of the master computer component of the invention.

Figure 12 depicts further detail of the get byte from compress buffer step of the down load image step of the master computer component of the invention.

Figure 13 depicts further detail of the write data to head buffer step of the down load image step of the master computer component of the invention.

Figure 14 depicts further detail of the flush head buffer step of the write data to head buffer step of the master computer component of the invention.

Figure 15 depicts a flow chart diagram of the slave component of the invention, which is a detailed flow diagram of the current preferred embodiment of the Slave computer process of the invention.

Detailed Description of the Invention

Figure 1 shows a computer network having a source (master) computer 101 and a number of destination (slave) computers 102a, 102b, 102c, 102d, 102e, 102n connected to

each other electronically as may be used in the peer-to-peer mode of operation of the invention. A typical stand-alone computer may include the following components: a processor; internal memory; a disk storage device; a display device; and an input device. The electrical connection 103 provides a communication channel between the computer systems. The use of this invention does not require that the connection between the computer systems, as designated 103, necessarily be electrical. Other alternative methods of conductivity include, fiber optical, RF, and/or light wave transmission and detection. Often these types of communications channels are referred to as "networks," "local-area-networks" (LANs), and "wide-area-networks" (WANs). Typically, each computer is capable of operating as a stand-alone system. With the addition of the electrical connection 103 the computers are also capable of sharing information (for example data files and e-mail). The applicants' present invention extends the capabilities of individual computer systems connected over a computer network by providing disk imaging from one computer to another computer over a network. Such disk imaging provides a solution to the problem of installing, backing up, maintaining, and upgrading of computer operating system software, computer system utility software, and application programs. This invention, which operates either with or without a computer network server, permits a single computer system to be designated as a "master" and to transfer data from its disk drive or drives to one, some or all of the other computers on the network, designated as "slaves."

Figure 2 depicts a system diagram of a computer network employing the traditional network server hardware 201. This system is well known for providing a method of transferring data to and from the network server and from and to the individual computers on the network. These prior approaches, typically are referred to as "client-server" systems. In a

traditional client-server system, data is transferred from a client computer 202 to the network server 201, or is transferred from the server 201 to one or more of the client computers. While applicants' invention supports client-server network systems, the most important achievement of this invention is that it provides the capability of transferring data, files, and disk sectors from any computer on the network to any other computer on the network with or without a network server, thereby dramatically decreasing the difficulty and complexity of computer network management. Where a network server 201 is employed in the network the "master" computer 202a is provided with the capability of transferring data either to the network server 201 and/or to one or all of the "slave" computers 202b, 202c, 202d, 202e, . . ., 202n. This inventions supports networks utilizing client/server modes of operation for compatibility with existing networks. It also supports and provides peer-to-peer operation to permit network management without the considerable expense of network server hardware and simultaneously providing a significant improvement in data transfer efficiency across a computer network.

In its best mode of operation, the present invention operates through coordinated use of two computer system communications programs: IMGBLSTR and IMGSLAVE. IMGBLSTR is designed to operate on the "master" computer, while IMGSLAVE is designed to operate on the "slave" computer. In the current preferred embodiment of the invention each program is written in the C programming language and operates on a DOS operating system. However, other equivalent embodiments of the invention may be created in other computer languages, including but not limited to C++, Pascal, and assembly code, and may be designed to operate on other computer operating systems, including but not limited to UNIX, Windows and MacIntosh. The following discussion of the steps of the process of this

invention correspond to programs, sub-programs, and routines in IMGBLSTR and IMGSLAVE best mode of the invention. Following this discussion of the essential steps of this process is a complete list of the source code for each program. This source code is provided as part of this disclosure, to provide a fully enabling description of the invention. It is suggested that the reader refer to this source code for additional detailed description as the reader reviews the following figure by figure discussion of the process of the invention.

The following description of the top level state diagrams of the "master" computer and the "slave" computer is provided to give the reader an overview of the handshaking and data processing process of the invention. A detailed breakdown of each subprocess, task or program is given in the later detailed descriptions of the process flow charts and the best mode of operation of the invention is provided in the listing of software source code included in this description following the description of the detailed subprocesses, tasks, and/or programs.

Figure 3 depicts the top level state diagram of the master computer ("IMGBLSTR") program / process component of the invention. The master computer process begins by entering the Wait for Slave Connection state 301. In this state 301 the master computer is initialized for communication with slave computers. The process goes from state 301 to state 302 when the master computer broadcasts its disk drive geometry to the slave computers. In this Send Geometry Packet state 302 the master computer sends the master computer drive geometry to slave computers on the send IPX socket. The "currently connected clients" display is initialized. Geometry packets are sent periodically to the slave computers. Geometry packets, in the preferred embodiment of the invention, include the following disk drive information for the master local disk drive: maximum cylinders; maximum heads;

maximum sectors, image file partition start cylinder; and image file partition end cylinder. Geometry packets also include information on the IPX network sockets, including: the network address; the node address; and the socket number. Error conditions are monitored and the master computer waits for RSVP packets from the slaves. When the master computer receives an RSVP from the slave the master adds the slave (or client) to its list and send an ACK packet to the slave 303. When the connection between the master and one or more slaves is either completed, terminated by an operator command, or terminated by a timeout, the master process goes to the Send Data Mode state 304. The Send Data Mode state 304, in combination with the Send Head Broadcast state 306 and the Loop to Send All Tracks state 305, performs the upload image, broadcast head, broadcast skip head, broadcast given head, and the send IPX packet tasks. In the event that a Resend Request is received the process goes from the Send Data Mode state 304 to the Queue Req. On Resend List - Perform Throttle Processing 307. During this state, incoming packets are processed, resend requests are processed by servicing the resend queue by sending data packets to slaves, resend requests are recorded, the resend list is processed and tracks are re-transmitted. These tasks are processed in conjunction with state 306. In the event that no more data remains to be sent, the process moves to the Allow Slaves to Catch-Up state 308. This state finishes processing of broadcasts and handles late resend requests. The transition of the process from state 308 to state 309 is accomplished as follows: (1) a timer indicates it is time to send a packet; (2) a packet of data is sent 309; and (3) the process returns automatically to state 308 to wait for the next timer indication. Thus, when the timer has run, the process moves from the Allow Slaves to Catch-Up state 308 to Send "End of Data" Packet state 309, which sends farewell broadcast packets, and back to state 308. When the transfer is complete (when a timeout

occurs), the process moves to the Farewell to Slaves state 310, which in combination with the Send "Goodbye" Broadcast state 312 and the Remove Slave From List Send "Goodbye" to Specific Slave state 311, performing the say Goodbye to Slaves and Processing Image Packets tasks.

Figure 4 depicts the top level state diagram of the slave computer ("IMGSLAVE") program / process component of the invention. Beginning with the Hook-Up with Master state 401 the slave computer attempts to form a communications link with the master computer. After a geometry packet is received from the master, the slave process moves to the Send RSVP state 402, where the slave checks the disk drive geometry for compatibility and acknowledges receipt of the geometry information with an RSVP. After the slave process receives a RSVP acknowledge from the master, the process enters the Download Data state 404. Data is received via transitioning between the Download Data state 404 and the Process Data Packet state 407. The Process Data Packet state 407 processes the received data by checking to see if the head is at a valid position, and storing the received data in a buffer. A test is performed to determine if the designated slot is not empty, wherein a "lost data" resend request is sent. The process of the invention provides the capability of sending resend requests through the Send Resend Request state 408. If a "flush" packet is received (a "flush" packet is a special type of data packet, containing an attribute to indicate that when the data is copied to the image file, that the image file should then be flushed to disk), the Flush Complete Track state 405 is entered, where if the buffer is complete, it is written to disk and the "good upto" track is updated, removing the track from the "missing list." If the buffer is incomplete, this state 405 sends a "lost data" resend request and adds the track to the "missing list." When the transfer of data is complete, a "End of Data" packet is received

from the master, leading the slave process to the Farewell to Master state 409, where once a "goodbye" is received from the master, the Random Delay - Send "Goodbye" to Master state 410 is entered to send a "goodbye" is send to the master. Thereby, finishing the slave process.

Figure 5 depicts the top level process flow chart of the master computer component of the invention. In its current best mode of operation, the process of this invention begins with the initialization 501 of the master. During this initialization step 501 the user license is verified, usage parameters are checked the MAC address of the computer is acquired and displayed, the local drive geometry is retrieved, all needed buffers are allocated, and the number of hard disk drives installed in the computer is determined. Next, the command line process step is performed 502, where the command line is parsed, setting flags and/or calling the functions to process each command. Next, menus are processed 503. The process menu step calls other functions to get needed information for the execution of the program. A test is made as to whether the process is required to upload data ("image") to a slave or download data ("image") 504. If the process is required to upload data then the upload image step 505 is performed to upload the image, drive data, to the image file. If the process is not required to upload data then the download image step 506 is performed, which accomplishes the downloading of the image (drive data). If a file name was specified by the operator, it is opened and the drive data is read from the file. Lastly, the clean-up step 507 is performed to reset all affected computer systems on the network.

Figure 6 depicts the detailed flow diagram of the process menus steps of the master computer component of the invention. The process menus program process calls other program and/or functions to get the needed information for the process of the invention.

Where the requested information has already been entered on the command line, the associated menu is not displayed. First, the program variables are initialized 601. Next, the desired process function is requested 601, allowing the operator to specify whether the operator wants to upload or download. Next, the get partition 603 step is performed to display the local drive partition table and to allow the operator to specify the partition to be imaged. The next step is to get the image file name 604, during which the operator is prompted to provide the name of the image file, including the complete path. Next, the broadcast state is requested 605, that is, whether the operator wants to send the drive data to other slave computers via broadcast packets.

Figure 7 depicts the detailed flow diagram of the upload image step of the master computer component of the invention. The upload image program process performs the function of uploading the image (drive data) from a master computer to a network server computer and/or, if the broadcast feature is enabled, broadcasting the data to the image slave computers on the network. Initially, upload image program is initialized 701. This initialization step is necessary to provide needed data. A test 712 is made to determine whether the invention is operating in the client/server mode. If it is, an image file is opened 702 and an image header is written 703. The drive head buffer is read 704. A test is performed to determine whether the data is to be broadcast 705 to multiple slave computers. If the data is to be broadcast the broadcast is performed 707, if not the process passes through the null state 706 before once again testing to determine if the process is in the client/server mode 713. If it is, the data is compressed and written to a file 708. Data compression is accomplished, currently by a "Run-Length Encoding" scheme well known in the art. Next, System pointers are incremented 709 and if the system has not reached the stop cylinder 710

the process continues to read the head buffer 704, broadcasts the data 707, compresses the data, and adjusts the system pointers. After the stop cylinder is reached 710, the process performs the clean-up step 711 by closing files, resending missed tracks and saying "goodbye" to slave (client) computers.

Figure 8 depicts further detail of the read head step of the upload image section of the master computer component of the invention. This program step reads a head worth of data from a selected local hard disk drive at the current cylinder and current head location. Errors that are encountered, are reported. The process of the invention is aborted after five retries on errors. However, because CRC type errors are recoverable, they will not count towards aborting the process. First, this sub-process is initialized 801, providing access to the necessary variables. Next, the data in the designated current head is read 802. If a read error occurs 803 the read of head data is attempted again, up to five times 805. If errors continue beyond five tries 805 an error message is displayed 806 and the program is exited 807. Otherwise, if the data is read without error then the process passes through the null state 804 and the program returns to the upload image process step 505 as shown in figure seven.

Figure 9 depicts further detail of the broadcast head step of the upload image section of the master computer component of the invention. The function of the broadcast head program - subprocess is to broadcast the drive head data (headbuffer) to the image slaves using the send IPX socket. Data is sent 512 bytes at a time, equal to a single sector of disk data. On the last sector sent, the command to flush the data, that is, to write it to the hard drive, is sent. The first step in the broadcast head process is to initialize the data for broadcast 901. The process of sending packets of data 905 is performed one sector at a time. After it is determined that an additional sector of data is in the head 902, the sector data is

packaged 905 or prepared for broadcast, a delay is provided 903 to insure correct interpacket data broadcast, then the packet is sent 905. The broadcast head process is finished when no further sectors remain for transfer.

Figure 10 depicts further detail of the download image step of the master computer component of the invention. This program sub-process of the invention performs the function of downloading the image (disk drive data). If a file name is specified by the operator, it is opened and the disk drive data is read from the file. If the broadcast feature is enabled, the data is broadcast to the image slaves on the network. The first step is to initialize 1001 the needed data and to set the head buffer pointer to 0. Next, an image file is opened 1002 if needed and if the image file name is valid. The image header data is read 1003 and a check is made for the proper drive geometry. If the image geometry is equal to the drive geometry 1004 then the data is decompressed 1006 using well known "Run-Length" decompressions schemes. Once the data is decompressed it is written 1007 to the head buffers. After data is written to the head buffer, this part of the process is complete. If the image geometry does not match the drive geometry, of step 1004, an error message 1005 is displayed.

Figure 11 depicts further detail of the fill compress buffer from file step of the down load image step of the master computer component of the invention. This function fills the compress buffer by reading a head size (maximum sectors times 512) of data from the image file. If an error occurs during the read, an error message is displayed. First, the data is initialized 1101 to provide data for the process. Next, the compress buffer file size is read 1102 and tested 1103 for error. If an error is encountered, it is displayed 1105, otherwise a null state 1104 is used and the process returns to the download image process of the invention

of figure 10.

Figure 12 depicts further detail of the get byte from compress buffer step of the down load image step of the master computer component of the invention. This process subprogram functions to retrieve a byte from the compress buffer. When the compress buffer is empty, it is refilled by reading data from the image file. First, the data is initialized 1201 for use in the process. Next the compress pointer is tested 1202 to determine if it is greater than the size of the compress buffer. If it is, the compress buffer is filled 1203 from the image file. The compress pointer is then reset 1204. The byte pointed to by the pointer/counter is returned 1206 and the process returns to the download image process of figure 10. In the event that the compress pointer does not exceed the size of a buffer, test step 1202, the process goes through a null state 1205 and returns 1206 the byte pointed to by the compress pointer and returns to the download image process.

Figure 13 depicts further detail of the write data to had buffer step of the down load image step of the master computer component of the invention. This step functions to take the byte of data passed to it and write it to the head buffer the number of times indicated by the compress counter. This is part of the process of decompressing the image file. When the head buffer is full, it flushes the data to the hard disk drive. First, the data required is initialized 1301. Next, the data is written into the head buffer 1302. The head buffer pointer is incremented 1303. A test 1304 is made to determine whether the head buffer is full. If it is, the head buffer is flushed to disk 1305 and the pointers/counters are adjusted 1306. A test 1308 is then made to determine whether the compress count is complete. If it is, the process returns to the download image of figure 10. If it is not, then the process returns to step 1302 to move a byte into the head buffer. If the head buffer, of step 1304, is not full, a null state

1307 is passed through before the compress count test of step 1308.

Figure 14 depicts further detail of the flush head buffer step of the write data to head buffer step of the master computer component of the invention. This step functions to flush the head buffer data to disk. The entire head buffer is written in one command. All needed pointers, counters, etc., are updated along with screen information. Also, if the broadcast feature is enabled, the head data is broadcast to the image slaves. First the data is initialized 1401 for use in this process. Next, a test 1402 is made to determine if the process is in the broadcast mode. If it is, the head data is broadcast 1403 to all image slave computers. If not, the process goes through a null state 1404. The head buffer is then written 1405 to disk and the current head and cylinder data is adjusted 1406 prior to returning to the write data to head buffer process of figure 13.

Figure 15 depicts a flow chart diagram of the slave component of the invention. is a detailed flow diagram of the current preferred embodiment of the Slave computer process of the invention. The IMGSLAVE is the slave component or process of the invention that provides the parallel disk image process. The slave uses an IPX socket to listen for data from the image master. A special header in the data from the master determines the function the slave will perform. The IMGSLAVE program cannot create or restore an image without the IMGBLSTR program. Its function is to listen to the network for data from the IMGBLSTR, to retrieve data from the network and to write it to the slave's local drive. The slave process begins by initializing 1501 data for processing. When a geometry packet is received 1502 from the master the slave responds with an RSVP 1503. Next, the slave listens for an RSVP acknowledge 1504 from the master. After the receipt of the RSVP acknowledge, a test 1505 is made to determine whether the register for download is valid, if not, a message indicating

that the transfer is unusable is sent 1506. Alternatively, if the register for download is valid, then the data is downloaded 1507 to the slave. Error checking 1508 is performed and errors are displayed 1509 if detected. If no errors are detected, a download complete message is displayed 1510.

The following is a listing of the computer source code which is the current best mode preferred embodiment of the invention. The reader can, by consulting this source code, learn all that is necessary to produce and use the invention.

Software Source Code

```
#define KEY_ENABLE_DEBUG  1        /* Causes the '@' key to toggle DebugBroadcast mode. */
//#define IGNORE_ORIGINAL_GEOMETRY      1        /* Turns on a Debug mode used to test
geometry independence. */
#undef IGNORE_ORIGINAL_GEOMETRY
/*****************************************************************************
 * (C) Copyright 1996-1997 KeyLabs, Inc.
 * All Rights Reserved.
 * This program is an unpublished copyrighted work which is proprietary
 * to KeyLabs, Inc. and contains confidential information that is not
 * to be reproduced or disclosed to any other person or entity without
 * prior written consent from KeyLabs, Inc. in each and every instance.
 * WARNING: Unauthorized reproduction of this program as well as
 * unauthorized preparation of derivative works based upon the
 * program or distribution of copies by sale, rental, lease or
 * lending are violations of federal copyright laws and state trade
 * secret laws, punishable by civil and criminal penalties.
 *****************************************************************************/
/*****************************************************************************
        Program:  IMGBLSTR.C
        Description:  This program is the master component of the parallel disk image process. This
master uses an IPX socket to send
data to the image slave.  A special header in this data determines the function the slave will perform.
An IPX socket is also used to receive requests from the slave (ie. re-send sector data).
There are five modes of operation for the master:
            1 - Create Image on network server.
            2 - Send image data to slave while doing #1.
            3 - Send image data to slave without doing #1.
            4 - Download image from network server.
            5 - Send image data to slave while doing #4.
```

```
        Author: Kevin J. Turpin        Date: 13 May 1996
    Mod 1: Added drive preparation option to write to all unused portions
        of a partition prior to uploading.        Kevin - Oct 16, 1996
    Mod 2: Added debug processing to display counters during upload.
        Kevin - Oct 29, 1996 (See DisplayStaticScreenData and
                                WriteDataToCompressBuffer functions)
    ************************************************************************/
    #include <stdio.h>
    #include <stdlib.h>
    #include <stddef.h>
    #include <stdarg.h>
    #include <conio.h>
    #include <dos.h>
    #include <dir.h>
    #include <alloc.h>
    #include <bios.h>
    #include <time.h>
    #include <string.h>
    #include <mem.h>
    #include <process.h>
    #include <io.h>
    #include <fcntl.h>
    #include <sys\stat.h>
    #include <ctype.h>
    #define NWDOS
    #include <nwipxspx.h>
    #include "largemsg.h"
    #include "freespc.h"
    #include "esr.h"
```

```
#include "imgslave.h"
#include "eval.h"
#include "license.h"
/*************************************************************************/
/*************************************************************************/
// The next #define's are for controlling how the program works for
// debug, normal.  If the #define DEBUG is uncommented, various debug
// messages will be displayed during execution.
//#define DEBUG                 1
// end of special #define's
/*************************************************************************/
/* Make sure this program has plenty of stack space
 * (the default is something like 4K).
 */
unsigned _stklen = 16384;
/*************************************************************************/
/* Space for the License information: */
#include "crypchar.h"       /* Get CRYPT...() macros and decrypt_printstring() proto. */
#define DISTRIB_BY_STRING           \
 CRYPT_16( 'D', 'i', 's', 't', 'r', 'i', 'b', 'u', 't', 'e', 'd', ' ', 'b', 'y', ':', ' ')
char distributedBy[] = {
        DISTRIB_BY_STRING,
        CRYPT_40( 'K', 'e', 'y', 'L', 'a', 'b', 's', ',', ' ', 'I',          \
                'n', 'c', '.', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', \
                ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', \
                ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),
0       /* NUL-terminator. */
};
/*************************************************************************/
```

```
/*************************************************************************/

#define TRUE   1

#define FALSE  0

#define WRITE  3

#define  READ  2

#define  EXIT   0

#define  STD_DATA     1

#define  FLUSH 2

#define  GEOMETRY    3

#define ENDOFFILE     0xFF

#define  HD_DRIVE     0x80

#define  COMPRESSIONKEY    0xD5

#define  FILLDATA          0xFE

#define IPXOPENSOCKETERR  -1

#define  ALLOCERR          -2

#define  BIOSERR           -3

#define  SECTORERR         -4

#define  BADGEOMETRY       -5

#define  FILEOPENERR       -6

#define READERR            -7

#define  VERIFYERR         -8

#define SENDPACKETERR      -9

#define WRITEERR           -10

#define CLOSEERR           -11

#define LICENSEERR         -12

#define NEEDFILENAMEERR   -13

#define  MAX_STATIC_SCREEN_DATA        29

#define  MAX_DYNAMIC_SCREEN_DATA       18

#define  LOCALMAC   0
```

```
#define  IMAGEFILE    1
#define  LOCALCYL     2
#define  LOCALHEAD    3
#define  LOCALSECT    4
#define  MASTERCYL    5
#define  MASTERHEAD   6
#define  MASTERSECT   7
#define  PROCCYL      8
#define  PROCHEAD     9
#define  PROCSECT     10
#define  SENDSOCKET   11
#define  RECVSOCKET   12
#define  STOPCYL      13
#define  STOPHEAD     14
#define  STOPSECT     15
#define  HEADSSENT    16
#define  HEADSMISSED  17
//------------------------- TYPES ---------------------------------
typedef struct client {
        struct client *prev;
        struct client *next;                    /* Doubly-linked list pointers. */
        BYTE   netAddress[ 12]; /* Client's IPX address. */
        /* Add other stuff here (e.g. throttling stuff) */
} ClientInfo;
//---------------------- DECLARATIONS ---------------------------
#include "prog_id.h"
static unsigned int ProgramID = IMAGEBLASTER_ID; /* From prog_id.h */
static unsigned int ProgramMajorVersion = 1;
static unsigned int ProgramMinorVersion = 4;
```

```
static unsigned int ProgramRevision = 11;
static int evalProgram = 0; /* Set when only license is eval. */
static int licenses = 0;/* Return value from licenseCount() -- used to determine
        * whether or not broacast stuff is enabled, and to
        * make sure users don't try to add more clients
        * than those for which the master is licensed.
        */
#define BROADCAST_ENABLED()     ( licenses > 1)
#define NUMBER_ECBs 20
ECB     receiveECB[ NUMBER_ECBs];
IPXHeader       receiveHeader[ NUMBER_ECBs];
BYTE    IPXDataBuffer[ NUMBER_ECBs][ 512 + sizeof(BufferPacketHeader)];
ClientInfo      *clients = NULL;
int     currentClientCount = 0;
int     expectedClientCount = -1;
WORD    LocalSocket;    /* Our dynamically-bound local socket. */
ECB     ReceiveECB,
        SendECB;
IPXHeader       SendHeader,
ReceiveHeader;
BYTE    NetAddress[12],
        IPXSendBuffer[512+22],
        IPXRecvBuffer[512+22];
int     i,
        value,
        HeadComplete = TRUE,
        ImageGeometryFound = FALSE,
        UploadFlag = FALSE,
        DownloadFlag = FALSE,
```

```
        AutomationFlag = FALSE,
        CommandLineFlag = FALSE,
        ImageFileNameFlag = FALSE,
        ImageFileNameValid = FALSE,
        BroadcastEnableFlag = FALSE,
        NoBroadcastEnableFlag = FALSE,
        ValidDataInHead = FALSE,
        ReadAfterWriteVerifyFlag = FALSE,
        DebugFlag = FALSE,
        DebugBroadcast = FALSE,
        DebugBroadcastDisplayMACs = FALSE,
        MonitorBroadcastDelays = FALSE,
        ForceWriteData = FALSE;
    int PrepareDriveFlag = FALSE;
char PrepareDriveLetter;/* Set when PrepareDriveFlag is set to TRUE. */
    int NTFlag = FALSE; /* Running on NT, so avoid physical disk calls. */
    int LocalMaxHeads,
        LocalMaxCylinders,
        LocalMaxSectors,
        LastSentSector = 0,
        PartitionNumber = -1,
        ImageFileHandle,
        DelayValue =0,
    //  DelayValue = 1,
        FlushDelay = 0,
    //  FlushDelay = 0,
        CurrentCylinder,
        CurrentHead,
        CurrentSector = 1,
```

```
        ValidUpThruCylinder = -1,
        ValidUpThruHead = -1,
        CompressCount = 1,
        NumberOfHardDrives = 1,
        CurrentHardDrive = 1,
        CurrentHDDrive = HD_DRIVE,
        HardDriveID[4] = {TRUE, FALSE, FALSE, FALSE};
unsigned long   totalHeadsSent = 0L;
unsigned long   totalHeadsMissed = 0L;
#define SPEEDUP_INTERVAL_SECONDS ( 20)
#define NEXT_SPEEDUP()( clock() + ( SPEEDUP_INTERVAL_SECONDS * CLK_TCK))
clock_t  timeForNextSpeedup = 0L;
enum {
        BLAST, CLEANUP
} Phase = BLAST;
#define WINDOW_SIZE  50
#define SLOWDOWN_THRESHOLD      5
unsigned int windowSamples[ WINDOW_SIZE];
int windowSlot = 0;
long numberSamplesThisWindow = 0L;
typedef struct {
        int     consecutiveDataLosses;
        long    headsSent;
        long    headsMissed;
} PerDelayCounts;
#define NUM_DELAYS ( 100)
PerDelayCounts  perDelayCounts[ NUM_DELAYS];
PerDelayCounts  outOfRangeCounts;
```

```
unsigned long precisionDelayLoopsPerSecond;/* For precise timing loops.
                    *(kind of like Linux BogoMips
* and jiffies).                                                    */
int     xcord, ycord;
long    CompressPTR = 0,            // for compression algorithm
        BytesInCompressBuffer,
        HeadBufferPTR = 0;
long    FillTimes=0;
long    CKCount = 1,                // Counters for Debug purposes
        FDCount = 1;
char    ErrorMessage[80],
        TmpBuffer[20],
        ImageFileName[80],
        FillData[4096];
unsigned char   * HeadBuffer,
                * CompareHeadBuffer,
                * CompressBuffer,
                LastData,           // for compression algorithm
                Current,
                ImageDataByte,
                TempBuffer[512],
                PartitionBuffer[512];
BufferPacketHeader OurHeader;
struct ImgFileHeader
{
//      int headerKey;
        int maxHeads;
        int     maxCylinders;
        int maxSectors;
```

```
//      long maxSectors;
        int partitionStartCylinder;
        int partitionEndCylinder;
        int compressionKey;
        int partitionNumber;
        unsigned char rawPartitionBuffer[512];
//      char imageFileName[80];
} ImageFileHeader;
struct PartitionTable
{
        int     startCylinder;
        int     endCylinder;
#if defined( IGNORE_ORIGINAL_GEOMETRY)
        int     startHead;
        int     endHead;
#endif
        int     bootFlag;
        int     type;
        long    totalSectors;
} Partition[5];                         // 5th entry is entire drive
struct DynamicScreenType
{
        int  x;
        int  y;
} DynamicScreenData[MAX_DYNAMIC_SCREEN_DATA] ={{ 24,9},// Local MAC addr

                                                { 52, 9},       //Image file
                                                { 24, 13},      //Local Cyl
                                                { 24, 14},      //Local Head
```

```
                                        { 24, 15},      //Local Sector
                                        { 64, 13},      //Image Cyl
                                        { 64, 14},      //Image Head
                                        { 64, 15},      //Image Sector
                                        { 29, 22-1}, //Proc Cyl
                                        { 40, 22-1}, //Proc head
                                        { 49, 22-1}, //Proc sector
                                        {  2, 22-1}, //Send socket
                                        {  8, 22-1}, //Recv socket
                                        { 29, 23-1}, //Stop cylinder
                                        { 40, 23-1}, //Stop Head
                                        { 49, 23-1}, //Stop Sector
                                        { 64+7, 22-1},// Heads Sent
                                        { 64+7, 23-1}// Heads missed
};
struct StaticScreenType
{
        int             broadcastRelated;
        int  x;
        int  y;
        char *  Text;
} StaticScreenData[MAX_STATIC_SCREEN_DATA]={{0,33,1, "I M G M A S T R"},
                                        {0,5,9, "Local MAC Address:"},
                                        {0,40,9, "Image File:"},
                                        {0,10,11,"Local Drive Geometry"},
                                        {0,50,11, "Image Drive Geometry"},
                                        {0,10,12, "--------------------"},
                                        {0,50,12, "--------------------"},
                                        {0,13,13, "Cylinders"},
```

```
{0,53,13, "Cylinders"},
{0,13,14, "Heads"},
{0,53,14, "Heads"},
{0,13,15, "Sectors"},
{0,53,15, "Sectors"},
{0,34,18-1, "Processing"},
{0,34,19-1, "----------"},
{0,27,20-1, "Cylinder"},
{0,27,21-1, "--------"},
{0,39,20-1, "Head"},
{0,39,21-1, "----"},
{0,47,20-1, "Sector"},
{0,47,21-1, "------"},
{1,4,18-1, "Sockets"},
{1,4,19-1, "-------"},
{1,2,20-1, "Send  Recv"},
{1,2,21-1, "----  ----"},
{1,64 + 7,20-1, " Heads"},
{1,64 + 7, 21-1, "------"},
{1,64,22-1, "  Sent:"},
{1,64,23-1, "Missed:"}
};
//------------------- Function Declarations --------------------------
void Initialize( void);
void GetDriveGeometry( void);
void ParseCommandLine(int argc, char * argv[]);
void ProcessSwitch(char * switchString);
void ProcessMenus( void);
void FunctionMenu( void);
```

```
void PartitionMenu( void);
void ImageFileNameMenu( void);
void BroadcastMenu( void);
//void SocketNumberMenu();
void ReadPartitionTable( void);
void ProcessImage( void);
void UploadImage( void);
void InitializeForBroadcast( void);
void DisplayProcessingScreen( void);
void BroadcastDriveGeometry( void);
void SendOurIPXPacket( void);
void SendOurIPXPacketSized( int dataSize);
void ReadDriveHead(unsigned char * buffer);
void BroadcastHead( void);
int CompressHead( void);
int WriteDataToCompressBuffer( void);
int FlushCompressBuffer( void);
void DownloadImage( void);
unsigned char GetByteFromCompressBuffer( void);
void FillCompressBufferFromFile( void);
void WriteDataToHeadBuffer(int compressCount, unsigned char data);
void FlushHeadBuffer( void);
void WriteHeadBufferToDrive( void);
void CheckGeometry( void);
void Beep( void);
void DisplayUsage( void);
void DisplayBiosError(int ccode, int x, int y);
void LogError(int errCode, char * errMessage);
void DisplayStaticScreenData( void);
```

```
void CleanExit( void);
void DisplayLocalGeometry( void);
void DisplayImageGeometry( void);
void DisplayProcData( void);
void DisplayLocalMAC( void);
void DisplaySocketNumbers( void);
void PrepareDriveWorker( char driveChar);
void PrepareDrive( void);
void DrawBarGraph(int PercentCompleted);
void WriteEvalMessage( void);
void DetermineNumberOfHardDrives( void);
void AllocateBuffers( void);
void DeallocateBuffers( void);
void finishBroadcastProcessing( void);
void handleLateResendRequests( void);
void sayGoodbyeToClients( void);
/*************************************************************************/
/* This"TextColor()interface should be used instead of textcolor() since
 * it returns the previous color, allowing code to RESTORE the color
 * to what it was before the change.
 */
static int currentTextColor = LIGHTGRAY;
int
Text_Color( int newColor)
{
        int oldColor = currentTextColor;
        textcolor( newColor);
        currentTextColor = newColor;
        return oldColor;
```

```
}
/* Catch uses of textcolor() from now on--should be using
 * Text_Color() instead.
 */
#define textcolor( a) HOSERMAMA_USE_Text_Color_INSTEAD_of_textcolor( a)
/*****************************************************************/
/* Inform at the top of the screen in a "rolling log window"
 * fashion.
 * Probably only called by DB_INFORM() and TEST_INFORM().
 *
 */
void
__INFORM( char *fmt, va_list ap)
{
        static int line = 1;
        int x = wherex(), y = wherey();
        int originalColor = Text_Color( LIGHTGRAY);
        gotoxy( 1, line);
        clreol();
        vprintf( fmt, ap);
        clreol(); /* Clear to end of "current" line. */
        line++;
        if ( line > 7) {
                line = 1;
        }
        gotoxy( 1, line);
        clreol(); /* Clear NEXT line. */
        gotoxy( x, y);      /* Restore original cursor. */
        Text_Color( originalColor);
```

```
}
/*************************************************************************/
/* "Debug Broadcast" inform.  Display the message ONLY IF the
 * DebugBroadcast flag is set.
 */
void
DB_INFORM( char *fmt, ...)
{
        if ( DebugBroadcast) {
                va_list ap;
                va_start( ap, fmt);
                __INFORM( fmt, ap);
                va_end( ap);
        }
}
/*************************************************************************/
/* Test inform.  Display the message.
 */
void
TEST_INFORM( char *fmt, ...)
{
        va_list ap;
        va_start( ap, fmt);
        __INFORM( fmt, ap);
        va_end( ap);
}
/*************************************************************************/
void
TopLargeMsg( int clearScreenFlag, char *msg)
```

```
{
        int x = wherex();
        int y = wherey();
        gotoxy( 1, 1);
        DisplayLargeMsg( clearScreenFlag, msg);
        gotoxy( x, y);
}
/*************************************************************************/
void
inform( FILE *stream, char *fmt, ...)
{
        va_list   ap;
        va_start( ap, fmt);
        vfprintf( stream, fmt, ap);
        va_end( ap);
}
/*************************************************************************/
void
calculatePrecisionDelay( void)
{
        /* These are volatile to try to get the compiler to NOT
         * optimize access to them, either by placing the variables
         * in registers or by completely optimizing-away references
         * to the variables.
         */
        volatile clock_t   end;
        volatile clock_t   now;
        volatile unsigned long loops = 0;
        DB_INFORM( "Calculating precision delay...");
```

```
        /*************************/
        /* Wait for tick edge */
        end = clock();
        while ( clock() == end)
        {}
        /*************************/
        /* Count one second. */
        end = clock() + CLK_TCK;
        while ( 1) {
                now = clock();
                if ( now > end)
                        break;
                loops++;
        }
        precisionDelayLoopsPerSecond = loops;
        DB_INFORM( "precisionDelayLoopsPerSecond is %lu\n", loops);
        if ( DebugBroadcast) {
                end = clock() + CLK_TCK;
                /* Pause so the numbers can be read. */
                while ( clock() < end)
                {}
        }
}
/*****************************************************************************/
void
tenthMilliDelay( unsigned int tenthMilliseconds)
{
        /* These are volatile to try to get the compiler to NOT
         * optimize access to them, either by placing the variables
```

```
 * in registers or by completely optimizing-away references
 * to the variables.
 */
volatile clock_t end;        /* DUMMY. */
volatile unsigned long loops = 0;
volatile unsigned long loop_limit;
if ( tenthMilliseconds == 0)          /* Short circuit. */
        return;
/* Calculate how many "loops" we need for the required delay. */
loop_limit = (  (unsigned long)tenthMilliseconds
                        * precisionDelayLoopsPerSecond
                        / 10000L);
/* If we're running on a machine that is SO SLOW that it can't
 * accurately meter out a small enough delay, just RETURN.
 * (The processor is so slow that the multiply/divide calculation
 * above probably took long enough. :-)
 */
if ( loop_limit == 0) {
        return;
}
/*This loop is coded in a somewhat strange manner in an attempt to
 * get the code path to match, as closely as possible, the delay-
 * calculation loop in calculatePrecisionDelay().
 */
while ( 1) {
        end = clock();     /* DUMMY, to match timing loop in calcPrecisionDelay... */
        if ( loops > loop_limit)
                break;
        loops++;
```

```
        }
    }
    /*************************************************************************/
    /* Display the current flush and interpacket delay values, but only
     * if the DebugBroadcast or MonitorBroadcastDelays variables are set.
     *
     */
    void
    showDelayValues( void)
    {
        static int labelsDone = 0;
        if ( DebugBroadcast || MonitorBroadcastDelays) {
            int x = wherex();
            int y = wherey();
            int originalColor = Text_Color(LIGHTGRAY);
            if ( ! labelsDone) {
                gotoxy( 55, 17);
                cprintf("     Flush Delay:");
                gotoxy( 55, 18);
                cprintf("Inter Delay (+/-):");
            }
            Text_Color( WHITE);
            gotoxy( 73, 17);
            cprintf("%4d", FlushDelay);
            gotoxy( 73, 18);
            cprintf("%4d", DelayValue);
            gotoxy( x, y);
            if ( ! labelsDone) {
                labelsDone++;
```

```
                }
                Text_Color( originalColor);
        }
}
/*************************************************************************/
void
displayTotalAndMissedHeads( void)
{
        if ( 0 == ( totalHeadsSent % 10)) {
                int x = wherex();
                int y = wherey();
                int originalColor = Text_Color( WHITE);
                gotoxy( DynamicScreenData[ HEADSSENT].x,
                        DynamicScreenData[ HEADSSENT].y);
                cprintf("%6lu", totalHeadsSent);
                gotoxy( DynamicScreenData[ HEADSMISSED].x,
                        DynamicScreenData[ HEADSMISSED].y);
                cprintf("%6lu", totalHeadsMissed);
                Text_Color( originalColor);
                gotoxy( x, y);
        }
}
/*************************************************************************/
/* Routines to compare one track to another. */
int
CHEqual( long cyl1, long head1,
                long cyl2, long head2)
{
        return (cyl1 == cyl2 && head1 == head2);
```

```
}
/*****************************************/
int
CHlessThan( long cyl1, long head1,
                long cyl2, long head2)
{
        return (  ( cyl1 < cyl2)
                || (  cyl1 == cyl2
                && head1 < head2));
}
/*************************************************************************/
//------------------ Start of program ------------------------------
int
main(int argc, char * argv[])
{
                /* Get license count before processing command-line
                 * options--some options are enabled or disabled
                 * based on the license count.
                 */
                licenses = licenseCount( NULL, ProgramID, ProgramMajorVersion);
                // check usage parameters, display USAGE if needed.
                if ((argc == 2) &&
                        ((strcmpi(argv[1], "?") == 0) ||
                         (strcmpi(argv[1], "/?") == 0))) {
                        DisplayUsage();
                        CleanExit();
                }
                // Parse the command line
                ParseCommandLine(argc, argv);
```

```
/* VERIFY license count after parsing command-line
 * options--this allows us to do things like display
 * the usage message and license info even though
 * the program isn't licensed.
 */
if ( licenses == 0) {
        printf("This program is not licensed.\n");
        exit( 1);
}
// Calculate number of precision loops per millisecond
// of delay
calculatePrecisionDelay();
if ( isEvalLicense( ProgramID, ProgramMajorVersion)) {
        evaluation_notice();        /* from common/client/eval/eval.c */
        evalProgram = 1;
}
if ( PrepareDriveFlag == TRUE) {
        /* User selected command-line option to prepare drive.
         * So Prep the drive and then exit.
         * Note that we do this BEFORE the call to Initialize()
         * so we can avoid the physical disk I/O associated with
         * getting drive geometry (this physical I/O is issued
         * in Initialize()).
         */
        PrepareDriveWorker( PrepareDriveLetter);
        CleanExit();
}
// Initialize the needed info for the program
Initialize();
```

```
                // Process the menus for data not on command line
                ProcessMenus();
                // Now process the image (either upload or download)
                ProcessImage();
                return 0;
}
/*************************************************************************
        Program:  Initialize
        Description: This function initializes the data for the program.It
                        does the following:
                        1 - Gets the local MAC address
                        2 - Gets the local drive geometry
                        3 - Allocates the buffers
        Author: Kevin J. Turpin         Date: 13 May  1996
        Modifications:
        Mod #1-Added call to DetermineNumberOfHardDrives.Kevin-Oct 30,1996
        Mod #2 - Added check for debug mode.  Kevin - Nov. 2, 1996
*************************************************************************/
void Initialize( void)
{
                // Are we in debug mode as per compile time?
                #ifdef DEBUG
                        DebugFlag = TRUE;
                #endif
                // Get the MAC address of this machine and display it
                IPXInitialize();
                IPXGetInternetworkAddress(NetAddress);
                // Get local drive geometry
                GetDriveGeometry();
```

```
                // Allocate all needed buffers
                AllocateBuffers();
                // Now that the buffers are allocated, lets determine how
                // many hard drives are in this computer.
                DetermineNumberOfHardDrives();
                // Let's read the local partition table
                ReadPartitionTable();
}
/*************************************************************************
        Program:  GetDriveGeometry
        Description: This function determines the local drive geometry by
                        reading the data from the drive partition table.
Author: Kevin J. Turpin         Date: 6 May  1996
*************************************************************************/
void GetDriveGeometry( void)
{
        int             ccode;
        ccode=biosdisk(8, CurrentHDDrive, 0, 1, 1, 1, TempBuffer);
        if (ccode) {
                sprintf(ErrorMessage, "\n\nBios error %x reading drive geometry.", ccode);
                LogError(BIOSERR, ErrorMessage);
                CleanExit();
        }
        LocalMaxCylinders = TempBuffer[1] + ((TempBuffer[0]>>6) << 8);
        LocalMaxSectors = TempBuffer[0] & 0x3f;
        LocalMaxHeads = TempBuffer[3];
        // Show Geometry if in debug mode
        if (DebugFlag == TRUE) {
                clrscr();
```

```
                printf("\nHeads = %d", LocalMaxHeads);
                printf("\nCylinders = %d", LocalMaxCylinders);
                printf("\nSectors = %d", LocalMaxSectors);
                printf("\n\nPress any key to continue .....");
                getch();
        }
}
/*************************************************************************
        Program:  ParseCommandLine
        Description: This function parses the command line, setting flags
                        and/or calling the functions to process each command.
                        See DisplayUsage() for a list and description of the
                        documented command-line switches.  See ProcessSwitch()
                        for information on UNDOCUMENTED switches.
        Author: Kevin J. Turpin         Date: 13 May  1996
*************************************************************************/
void ParseCommandLine(int argc, char * argv[])
{
        int i;
        char tmpBuf[80];
        for (i=0; i<argc; i++) {
                // Parse the switches
                strcpy(tmpBuf, argv[i]);
                if (tmpBuf[0] == '-') {                         // we have a switch
                        ProcessSwitch(tmpBuf);
                }
        }
}
/*************************************************************************
```

```
Program:  ProcessSwitch

Description:  This function process the switch string.

              See DisplayUsage() for a list of documented command-line
              switches.

              The UNDOCUMENTED switches are:

              -db     Turns on "debug broadcast" mode

              -db!    Turns on "debug broadcast" mode AND displays MAC
                      addresses of clients that request retransmissions.

              -M      Turns on "monitor broadcast" mode.  This mode shows
the delay values but does not print the othermessages.

              -Z!     Turns on DebugFlag (debug mode).

              -F      Turns on ForceWriteData mode.

Author: Kevin J. Turpin        Date: 13 May  1996

Mod #1 - Added debug switch key "!". Kevin - Nov. 2, 1996

Mod #2-Added the "R" switch for read after write verify.Kevin 12-17-96

Mod #3 - Added the "db" (debug broadcast) switch. Chris 12-27-1996.

**********************************************************************/

void ProcessSwitch(char * switchString)

{

    /* First check for multi-character switches. */

    /* NB: The "-db" switch is NOT documented in the 'usage' message. */

    if ( stricmp( "-db", switchString) == 0) {

        DebugBroadcast = TRUE;

        return;

    }

    /* Check for "verbose" debug mode, in which we display the MAC

     * addresses of clients that request retransmissions.

     */

    if ( stricmp( "-db!", switchString) == 0) {
```

```
                DebugBroadcast = TRUE;
                DebugBroadcastDisplayMACs = TRUE;
                return;
        }
        /* None matched, so check for single-character switches. */
                switch(toupper(switchString[1])) {
                        case 'U':       //Upload function
                        UploadFlag = TRUE;
                        break;
                        case 'D':       //Download function or debug
                        DownloadFlag = TRUE;
                        break;
                        case 'P':       //Partition number
                                PartitionNumber = atoi(&switchString[2]) - 1;
                                break;
                        case 'I':       //Image file name
                                strcpy(ImageFileName, &switchString[2]);
                                ImageFileNameFlag = TRUE;
                                if ( strlen( ImageFileName) > 0) {
                                        ImageFileNameValid = TRUE;
                                }
                                break;
                        case 'R':       //Read after write verify
                                ReadAfterWriteVerifyFlag = TRUE;
                                break;
                        case 'B':       //Broadcast enabler
                        if ( BROADCAST_ENABLED()) {
                                BroadcastEnableFlag = TRUE;
                                if ( isdigit( switchString[ 2])) {
```

```
int param;
if (sscanf(&switchString[2],"%u",&param) == 1) {
        if ( param == 0) {
        /* Switch -b0 says DISABLE broadcast. */
        BroadcastEnableFlag = FALSE;
        NoBroadcastEnableFlag = TRUE;
}
else {
expectedClientCount = param;
                                }
                        }
                        else {
printf("\n\nSwitch '%s' is invalid.", switchString);
CleanExit();
}
}
}
else {
printf("\n\nSwitch '%s' is invalid.", switchString);
CleanExit();
}
break;
case 'L':
displayMyLicenses();
CleanExit();
break;
case 'G': /* "G"et Ready (prepare drive). */
{
char driveChar = toupper( switchString[ 2]);
```

```
                    if ( 'A' <= driveChar && driveChar <= 'Z') {
                    PrepareDriveFlag = TRUE;
                    PrepareDriveLetter = driveChar;
                    }
                    else {
                    printf("\n\nInvalid drive letter.Valid range is 'A'..'Z'.\n");
                    CleanExit();
                }
            }
            break;
    case 'N': /* Option -NT */
    if ( toupper( switchString[ 2]) == 'T') {
            NTFlag = TRUE;
    }
    else {
            printf("\n\nSwitch -%s is invalid.", &switchString[1]);
            CleanExit();
    }
    break;
    /*************************/
    /* UNDOCUMENTED switches: */
    case 'M':
            MonitorBroadcastDelays = TRUE;// Shows broadcast delays.
            break;
    case 'Z':
            if (switchString[2] == '!')                                    // special
    debug flag key
                    DebugFlag = TRUE;
            break;
```

```
case 'F':
        ForceWriteData = TRUE;
        break;
default:
        printf("\n\nSwitch -%c is invalid.", switchString[1]);
        CleanExit();
        break;
}
}
/*************************************************************************
        Program:  ProcessMenus
        Description: This function calls other functions to get the needed
                        information for the program.  If the information has
                        already been entered on the command line (indicated by
                        flags), the associated menu is not displayed.
        Author: Kevin J. Turpin        Date: 13 May  1996
*************************************************************************/
void ProcessMenus( void)
{
        if (UploadFlag == FALSE && DownloadFlag == FALSE)
                FunctionMenu();
        if (PartitionNumber < 0)
                PartitionMenu();
        if (ImageFileNameFlag == FALSE)
                ImageFileNameMenu();
        if ( BROADCAST_ENABLED()) {
        if (BroadcastEnableFlag == FALSE && NoBroadcastEnableFlag ==FALSE)
                BroadcastMenu();
        }
```

```
        if ( ImageFileNameValid == FALSE && BroadcastEnableFlag == FALSE) {    /* No work to do!
*/
        sprintf(ErrorMessage, "\nNothing to do");
        DisplayLargeMsg(1, "  Error");
        LogError(FILEOPENERR, ErrorMessage);
        CleanExit();
        }
//      if ((SendIPXSocketNumber == 0  || RecvIPXSocketNumber == 0) &&
//              (BroadcastEnableFlag == TRUE))
//      SocketNumberMenu();
}
/*************************************************************************
        Program:  FunctionMenu
        Description: This function dipslays the function menu to allow the
        operator to specify whether he wants an upload or download operation.
        Author: Kevin J. Turpin        Date: 13 May  1996
*************************************************************************/
void FunctionMenu( void)
{
        char ch;
        do {
                clrscr();
                DisplayLargeMsg(1, " FUNCTION");
                printf("\n                    Function Menu");
                printf("\n                    ----------\n");
                printf("\n                    (U)pload Image");
                printf("\n                    (D)ownload Image\n");
                printf("\n              (P)repare drive for upload\n");
                printf("\n                 (Q)uit Program\n");
```

```
                printf("\n          Enter Function Option [U/D/P/Q]:");
                // Eval message
                WriteEvalMessage();
                ch = getch();
                if (toupper(ch) == 'U')
                        UploadFlag = TRUE;
                else if (toupper(ch) == 'D')
                        DownloadFlag = TRUE;
                else if (toupper(ch) == 'P')
                        PrepareDrive();
                else if (toupper(ch) == 'Q')
                        exit(0);
                else
                        Beep();
        } while (UploadFlag == FALSE && DownloadFlag == FALSE);
/*************************************************************************
        Program:  PartitionMenu
        Description:This function dipslays the local drive partition table
                        and allows the operator to specify the partition to be
                        imaged.
        Author: Kevin J. Turpin         Date: 13 May  1996
*************************************************************************/
void PartitionMenu( void)
{
        int i,
                number;
char    tmpChars[2];
static const struct StaticScreenTypeII
{
```

```
        int  x;
        int  y;
        char * text;
} staticScreenData[15] = {{ 33,  9, "Partition Menu"},
                          { 33, 10, "--------------"},
                          { 34, 12, "Cylinder"},
                          { 57, 12, "Total"},
                          { 68, 12, "Partition"},
                          {7,13, "Partition Active Status Start End Sectors"},
                          {7,14, "--------- ------------- ----- --- -------"},  { 57, 13,
"Bytes     Type"},
                          { 54, 14, "-----------  ----------"},
                          { 11, 15, "1"},
                          { 11, 16, "2"},
                          { 11, 17, "3"},
                          { 11, 18, "4"},
                          { 11, 20, "5     Entire Disk"},
                          { 11, 22, "Enter Partition Number [0=Exit] :"} };
int originalColor = Text_Color( LIGHTGRAY);
PartitionNumber = -1;
        do {
                clrscr();
                DisplayLargeMsg(1, " PARTITION");
                // Now display the static portion
                for (i=0; i<15; i++) {
                        gotoxy(staticScreenData[i].x,
                                staticScreenData[i].y);
                        cprintf("%s", staticScreenData[i].text);
                }
```

```
                    // If there is more than one drive in the system, allow user
                    // to switch between drives.
    if (NumberOfHardDrives > 1) {
            gotoxy(11,21);
            cprintf("9     Change Drives");
            Text_Color(WHITE);
            gotoxy(35,11);
            cprintf("Drive # %d", CurrentHardDrive);
    }
            // Now display the dynamic data and get option
    Text_Color(WHITE);
    for (i=0; i<4; i++) {
            if (Partition[i].totalSectors > 0) {
                    gotoxy(18, 15+i);
                    if (Partition[i].bootFlag == TRUE)
                            cprintf("Bootable");
                    else
                            cprintf("Nonbootable");
                    gotoxy(35, 15+i);
                    cprintf("%#3d %#3d %#7ld %#11ld",Partition[i].startCylinder,
                                    Partition[i].endCylinder,
                                                    Partition[i].totalSectors,
                                            Partition[i].totalSectors*512);
    // Display partition type
    gotoxy(67, 15+i);
    switch(Partition[i].type) {
            case 0x00:
                    cprintf("Unknown");
                    break;
```

```
                case 0x02:
                        cprintf("PCIX");
                        break;
                case 0x01:
                case 0x04:
                case 0x06:
                case 0x016:
                        cprintf("DOS, FAT");
                        break;
                case 0x05:
                        cprintf("Extended");
                        break;
                case 0x07:
                        cprintf("OS/2 HPFS");
                        break;
                case 0x0A:
                        cprintf("Boot Manager");
                        break;
                default:
                        break;
                        }
                }
        }
        // Write Eval only copy
        WriteEvalMessage();
        Text_Color(LIGHTGRAY);                  // change back to normal text
        // Now get the option
        gotoxy(45, 22);
        tmpChars[0] = getch();
```

```
tmpChars[1] = NULL;
number = atoi(tmpChars);
if (number == 0)
        CleanExit();
//if (number == 9 && NumberOfHardDrives == 2){//changedriveonlyif avail.
        if (number == 9) {
//      if (CurrentHardDrive == 1) {// currently 1st drive, make it 2nd
//              CurrentHardDrive = 2;
//              CurrentHDDrive = HardDriveID[1];// we found the id earlier
//      }
//      else {
//              CurrentHardDrive = 1;
//              CurrentHDDrive = HardDriveID[0];
//      }
        if (++CurrentHardDrive > 4)
                CurrentHardDrive = 1;
                while (HardDriveID[CurrentHardDrive-1] != TRUE) {
                if (++CurrentHardDrive > 4)
                CurrentHardDrive = 1;
        }
                CurrentHDDrive = 0x80 + CurrentHardDrive-1;
                GetDriveGeometry();     // get it for the new drive
                ReadPartitionTable();
                // We will need to re-do our buffers due to size changes
                DeallocateBuffers();
                AllocateBuffers();
        }
        else if (number > 0 && number < 6)
                if (Partition[number-1].totalSectors > 0 || number == 5)
```

```
                PartitionNumber = number-1;
            else
                Beep();
        } while (PartitionNumber == -1);
        Text_Color(originalColor);          // change back to normal text
}
/*************************************************************************
        Program:  ImageFileNameMenu
        Description:This function prompts the operator for the name of the
                    image file.  Name includes complete path.
        Author: Kevin J. Turpin     Date: 13 May  1996
*************************************************************************/
void ImageFileNameMenu( void)
{
        clrscr();
        DisplayLargeMsg(1, "  FILENAME");
        printf("\n                Image Filename Menu");
        printf("\n                ----------------\n");
        do {
            if (DownloadFlag == TRUE)// must enter file name if download
                printf("\nEnter image filename (complete path) : ");
            else
printf("\nEnter image filename (complete path) [Enter=No file] : ");
        // Write Eval only copy
        WriteEvalMessage();
        gets(ImageFileName);
        if (strlen(ImageFileName) > 0 ) {
            ImageFileNameValid = TRUE;
        }
```

```
            if ( UploadFlag == FALSE && ImageFileNameValid == FALSE) {
                    printf("\n"
                            "\n"
                            "Filename REQUIRED for download.\n"
                            "\n");
            }
            /* We REQUIRE a filename for downloads. */
    } while ( UploadFlag == FALSE && ImageFileNameValid == FALSE);
    }
/*************************************************************************
            Program:  BroadcastMenu
            Description:This function determines whether the operator wants to
                            send the drive data to other slave computers via
                            broadcast packets.
            Author: Kevin J. Turpin         Date: 13 May  1996
*************************************************************************/
    void BroadcastMenu( void)
    {
                    clrscr();
                    DisplayLargeMsg(1, " BROADCAST");
                    printf("\n                Broadcast Mode Menu");
                    printf("\n                ----------\n");
            printf("\nThe drive data can be broadcast to other computers                running the");
            printf("\nIMGSLAVE program.");
            printf("\nDo you want to use this broadcast feature? (Y/N)[N]: ");
            // Write Eval only copy
            WriteEvalMessage();
            if (toupper(getch()) == 'Y')
                    BroadcastEnableFlag = TRUE;
```

```
        else
                BroadcastEnableFlag = FALSE;
    }
    /*************************************************************************
        Program:  SocketNumberMenu
        Description: This function allows the operator to enter IPX socket
                     numbers for both the send and receive IPX sockets used
                     in the broadcast of drive data to the slaves.
        Author: Kevin J. Turpin        Date: 13 May  1996
    *************************************************************************/
    #if 0
    void SocketNumberMenu()
    {
    char tmpString[20];
    printf("\n\nTwo IPX sockets are used for thebroadcastingof drivedata.");
    printf("\nYou may specify the socket numbers for these two sockets.");
    printf("\nTo avoidconflict withother applications on the network, the");
    printf("\nsocket numbers should be between aaaa and bbbb (hex).");
    printf("\n\nThe sendsocket numbermust matchthat usedas receive socket");
    printf("\nnumber by the IMGSLAVE slaves.");
    printf("\n\nEnter the send socket number [1984] : ");
    flushall();
        // Write Eval only copy
    WriteEvalMessage();
    gets(tmpString);
    if (strlen(tmpString) > 2)
        sscanf(tmpString, "%x", &SendIPXSocketNumber);
    else
        SendIPXSocketNumber = 0x1984;
```

```
printf("\n\nEnter the receive socket number [1985] : ");
gets(tmpString);
if (strlen(tmpString) > 2)
        sscanf(tmpString, "%x", &RecvIPXSocketNumber);
        else
                RecvIPXSocketNumber = 0x1985;
}
#endif
/*************************************************************************
        Program:  ReadPartitionTable
        Description:This functionread thepartition tableat thefront of the
                        local harddrive and parsesout the information for each
                        partition.
        Author: Kevin J. Turpin        Date: 13 May  1996
*************************************************************************/
void ReadPartitionTable( void)
{
        int ccode,
                PBOffset[4] = {446, 462, 478, 494};
        // Read the partition table at the start of the drive
        ccode = biosdisk(READ,          // read function
        CurrentHDDrive,                 // on the local drive
        0,                                      // head 0
        0,                                      // cylinder 0
        1,                                      // sector 1
        1,                                      // read 1 sector of data
        PartitionBuffer);               // put it in the Partition buffer
if (ccode) {
sprintf(ErrorMessage, "\n\nBios error %x reading partitiontable",ccode);
```

```
LogError(BIOSERR, ErrorMessage);
CleanExit();
}
// Now parse out the partition information
for (i=0; i<4; i++) {
        Partition[i].startCylinder = PartitionBuffer[PBOffset[i]+3] +
                                ((PartitionBuffer[PBOffset[i]+2]>>6)<<8);
        Partition[i].endCylinder = PartitionBuffer[PBOffset[i]+7] +
                                ((PartitionBuffer[PBOffset[i]+6]>>6)<<8);
#if defined( IGNORE_ORIGINAL_GEOMETRY)
        Partition[i].startHead = PartitionBuffer[PBOffset[i]+1];
        Partition[i].endHead = PartitionBuffer[PBOffset[i]+5];
        inform( stderr, "Partition %d: starthead %u, endHead %u\n", i,
                Partition[ i].startHead,
                Partition[ i].endHead);
                (void)getch();
#endif
        // the sector data is packed
memmove(&Partition[i].totalSectors,&PartitionBuffer[PBOffset[i]+12],4);
        Partition[i].bootFlag = FALSE;
        if (PartitionBuffer[PBOffset[i]] == 0x80)
                Partition[i].bootFlag = TRUE;
        Partition[i].type = PartitionBuffer[PBOffset[i]+4];
}
// Partition 5 will be the entire drive
Partition[4].startCylinder = 0;
Partition[4].endCylinder = LocalMaxCylinders;
// Save the partition data in raw format for the image file header
memmove(ImageFileHeader.rawPartitionBuffer, PartitionBuffer, 512);
```

```
}
/*************************************************************************
        Program:  ProcessImage
        Description: This function is the controlling function for the image      process. It determines
        whether an upload or download
        should be performed and calls the appropriate function.
        It also governs the cleanup if the broadcast feature
        is used.
        Author: Kevin J. Turpin        Date: 14 May  1996
*************************************************************************/
void ProcessImage( void)
{
        // We need to open some sockets and so forth if broadcast
        if (BroadcastEnableFlag == TRUE)
                InitializeForBroadcast();
        // Now lets process the image
        if (UploadFlag == TRUE)
                UploadImage();
        else
                DownloadImage();
}
/*************************************************************************
        Program:  UploadImage
        Description: This function performs the uploading of the image
                (drive data). If a filename was specified by the operator,
                it is    opened and the drive data is written to the file.
                        If the broadcast feature is enabled, the data is broadcasted
                        to the image slaves on the network.
```

```
        Author: Kevin J. Turpin        Date: 14 May 1996
        ************************************************************************/
    void UploadImage( void)
    {
        // Initialize needed data
        // Open the image file if needed and write the image header.
    //          ImageFileHeader.headerKey = 0xaa;
        ImageFileHeader.maxHeads = LocalMaxHeads;
        ImageFileHeader.maxCylinders = LocalMaxCylinders;
        ImageFileHeader.maxSectors = LocalMaxSectors;
        ImageFileHeader.partitionNumber = PartitionNumber+1;      //make 1-5
        ImageFileHeader.partitionStartCylinder =
            Partition[PartitionNumber].startCylinder;
        ImageFileHeader.partitionEndCylinder =
            Partition[PartitionNumber].endCylinder;
        ImageFileHeader.compressionKey = 0xd5;
    //          strcpy(ImageFileHeader.imageFileName, ImageFileName);
        if (ImageFileNameValid == TRUE) {
            if ((ImageFileHandle = open(ImageFileName,
                                        O_WRONLY|O_CREAT|O_BINARY|O_TRUNC,
                                        S_IWRITE)) == -1) {
            sprintf(ErrorMessage, "\nError opening image file %s",
                ImageFileName);
            DisplayLargeMsg(1, "  Error");
            LogError(FILEOPENERR, ErrorMessage);
            CleanExit();
                }
            if ( write(ImageFileHandle, &ImageFileHeader,
```

```
                    sizeof(ImageFileHeader))
                     != sizeof( ImageFileHeader)) {
            sprintf(ErrorMessage, "\nError writing image file %s", ImageFileName);
                    DisplayLargeMsg(1, "  Error");
                    LogError(WRITEERR, ErrorMessage);
                    CleanExit();
                    }
            }
            // If broadcast enabled, lets pause and let operator start process
    if (BroadcastEnableFlag == TRUE){
            DisplayLargeMsg(1, "  PAUSING");
            WriteEvalMessage();
            printf("\n\nPress any key to start the upload and broadcast
                    process...");
                    BroadcastDriveGeometry();// so slaves can determine if image fits
            }
    // Now process the desired part of the drive.
            DisplayProcessingScreen();
            for (CurrentCylinder = Partition[PartitionNumber].startCylinder;
                    CurrentCylinder <= Partition[PartitionNumber].endCylinder;
                    CurrentCylinder++) {
    #if defined( IGNORE_ORIGINAL_GEOMETRY)
            for (CurrentHead = ( ( CurrentCylinder == Partition[ PartitionNumber].startCylinder)
            ? Partition[ PartitionNumber].startHead
            : 0);
    #else
            for (CurrentHead = 0;
    #endif
            CurrentHead <= LocalMaxHeads;
```

```
        CurrentHead++) {
        DisplayProcData();              // display where we are
        ReadDriveHead(HeadBuffer);      // read this head
        /* Update "valid up to" numbers. */
        ValidUpThruCylinder = CurrentCylinder;
        ValidUpThruHead = CurrentHead;
        if ( CompressHead()) {          // compress what we can
                sprintf(ErrorMessage, "\nError writing image file %s",
                        ImageFileName);
        DisplayLargeMsg(1, "  Error");
        LogError(WRITEERR, ErrorMessage);
        CleanExit();
        }
        if (BroadcastEnableFlag == TRUE) {
                BroadcastHead();
                ValidDataInHead = FALSE;
        }
        if ( BroadcastEnableFlag != TRUE) {
                if (kbhit()) {
                if (toupper(getch()) == 'Q')
                CleanExit();
        }
    }
    }
}
#if 0   /* Code moved to finishBroadcastProcessing(), which is
        * called below.
        */
// If in broadcast mode, send exit command
```

```
if (BroadcastEnableFlag == TRUE) {
        OurHeader.command = EXIT;
        for (i=0; i<3; i++)          // multiple times so were sure they get it
                SendOurIPXPacket();
        }
#endif
        if (CompressCount > 0) {// If we were compressing before
        if ( FlushCompressBuffer()) {       // move it to the CompressBuffer
                /* ERROR writing to disk. */
        sprintf(ErrorMessage, "\nError flushing buffer to image file %s",
                ImageFileName);
                DisplayLargeMsg(1, "  Error");
                LogError(WRITEERR, ErrorMessage);
                CleanExit();
                }
        }
        // Close the image file if used
        if (ImageFileNameValid == TRUE) {
                if ( close(ImageFileHandle)) {
        sprintf(ErrorMessage, "\nError closing image file %s",
                ImageFileName);
                DisplayLargeMsg(1, "  Error");
                LogError(CLOSEERR, ErrorMessage);
                CleanExit();
                }
        }
        /*****************************************/
        if ( BroadcastEnableFlag) {
        finishBroadcastProcessing();        /* Resend any missed tracks, then
```

```
                * say "goodbye" to the clients.
                */
        }
    }
/*************************************************************************
        Program:  InitializeForBroadcast
        Description: This function prepares for the broadcasting of data.
                It initializes IPX and opens the send and receive IPX
                sockets.
        Author: Kevin J. Turpin        Date: 14 May 1996
*************************************************************************/
void InitializeForBroadcast( void)
{
        /* Open Sockets */
        LocalSocket = 0; /* Request DYNAMIC socket. */
        if (IPXOpenSocket((BYTE far *)&LocalSocket, 0) != 0) {
                sprintf(ErrorMessage, "\n\nError opening dynamic local socket!");
                LogError(IPXOPENSOCKETERR, ErrorMessage);
                CleanExit();
        }
        DB_INFORM( "Opened DYNAMIC socket 0x%x\n", SWAP_WORD( LocalSocket));
        /*****************************************/
        /* Initialize "completed ECBs" list. */
        initEsrQueue();
        /* Initialize and Post a bunch of receive ECBs */
        for (i=0; i<NUMBER_ECBs; i++) {
                receiveECB[i].ESRAddress = ESRHandler;
                receiveECB[i].socketNumber = LocalSocket;
                receiveECB[i].fragmentCount = 2;
```

```
                receiveECB[i].fragmentDescriptor[0].address =
                        &receiveHeader[i];
                        receiveECB[i].fragmentDescriptor[0].size =
                        sizeof(IPXHeader);
                        receiveECB[i].fragmentDescriptor[1].address =
                        IPXDataBuffer[i];
                        receiveECB[i].fragmentDescriptor[1].size = 512 + sizeof(
                        BufferPacketHeader);
                        IPXListenForPacket( &receiveECB[ i]);
                }
                }
                /*********************************************************************/
                /* Cancel any pending IPX listens. */
                void
                ShutdownBroadcast( void)
                {
                        int i;
                        /* CANCEL any pending listens. */
        for (i=0; i<NUMBER_ECBs; i++) {
                if ( receiveECB[ i].inUseFlag) {
                        if ( IPXCancelEvent( &receiveECB[ i])) {
                                inform( stderr, "\nFailed to cancel IPX listen.\n");
                        }
                }
        }
}
/*********************************************************************

        Program: DisplayProcessingScreen

        Description:  This function displays the static portion of the
```

```
                processing screen;

        Author: Kevin J. Turpin        Date: 14 May 1996
*************************************************************************/
void DisplayProcessingScreen( void)
{
        int originalColor = Text_Color(LIGHTGRAY); clrscr();
        DisplayLargeMsg(1, "IMGBLASTER");
        // Display the static screen text
        DisplayStaticScreenData();
        DisplayLocalMAC();
        DisplayLocalGeometry();
        DisplayImageGeometry();
        DisplaySocketNumbers();
                // Write Eval only copy
        WriteEvalMessage();
        Text_Color( originalColor);
}
/*************************************************************************/
/* Display the list of connected clients. */
void
DisplayClients( void)
{
        int x = wherex(), y = wherey();
        ClientInfo *c;
        gotoxy( 1, 12);
        cprintf("Connected Slaves: Currently: %d",
                currentClientCount);
        if ( expectedClientCount != -1) {
                cprintf("  Expecting: %d",
```

```
                        expectedClientCount);
    }
        cprintf("\r\n");
    if ( DebugBroadcast) {
        /* Show the MAC addresses for each connected client. */
        for ( c = clients ; c != NULL ; c = c->next) {
            cprintf("%02x%02x%02x%02x%02x%02x ",
                c->netAddress[ 4],
                c->netAddress[ 5],
                c->netAddress[ 6],
                c->netAddress[ 7],
                c->netAddress[ 8],
                c->netAddress[ 9]);
            }
        }
        gotoxy( x, y);
    }
    /*************************************************************************/
    /* Add information about a client to the client list.
     * If the client is already on the list, just return success.
     *
     * Return ZERO on success, nonZero on failure.
     */
    int
    addClientToList( ECB far *ecb)
    {
        ClientInfo      *c;
        IPXHeader *hdr;
        hdr = (IPXHeader *)ecb->fragmentDescriptor[ 0].address;
```

```
        for ( c = clients ; c != NULL ; c = c->next) {
                if ( memcmp( c->netAddress, &hdr->source, 12) == 0) {
                        /* Client already on the list. */
                        return 0; /* Success. */
                }
        }
        /* Client not in the list: add a new node. */
        c = (ClientInfo *)malloc( sizeof( ClientInfo));
        if ( c == NULL) {
                return 1; /* Error. */
        }
        memcpy(c->netAddress, &hdr->source, 12);
        c->prev = c->next = NULL;
        if ( clients) {
                clients->prev = c;
                c->next = clients;
        }
        clients = c;
        currentClientCount++;
        DisplayClients();
        return 0;
}
/*********************************************************************/
/* Send an RSVP ACK packet to a particular client.
 *
 * Return ZERO if successful, nonZero otherwise.
 */
int
sendRSVPAck( ECB far *ecb)
```

```
{
        ECB     sendECB;
        IPXHeader header;
        RSVPPacket      rsvp;
        int transportTime;
        IPXHeader *reqPkt = (IPXHeader *)ecb->fragmentDescriptor[ 0].address;
/* Fill in the RSVP packet data
 */
rsvp.command = CONFIRM_DOWNLOAD;
/* Fill in the IPX packet header */
memmove( &header.destination,
        &reqPkt->source,
        sizeof( reqPkt->source));
        header.packetType = 4;
       /* Fill in the ECB. */
       sendECB.ESRAddress = 0;
       sendECB.socketNumber = LocalSocket;
       sendECB.fragmentCount = 2;
       sendECB.fragmentDescriptor[0].address = &header;
       sendECB.fragmentDescriptor[0].size = sizeof( header);
       sendECB.fragmentDescriptor[1].address = &rsvp;
       sendECB.fragmentDescriptor[1].size = sizeof( rsvp);
       if ( IPXGetLocalTarget( (BYTE far *)&reqPkt->source,
               sendECB.immediateAddress,
               &transportTime)) {
               return 0xff;        /* Error. */
       }
       /* Send the packet. */
       IPXSendPacket( &sendECB);
```

```
        while ( sendECB.inUseFlag) {
                IPXRelinquishControl();
        }
        return sendECB.completionCode;
}
/*********************************************************************/
/* Send a Geometry broadcast packet.
 *
 * Return ZERO if successful, nonZero otherwise.
 */
int
sendGeometryBroadcastPacket( void)
{
        ECB     sendECB;
        IPXHeader       header;
        GeometryPacket  geomPacket;
        /* Fill in the packet data
         */
        geomPacket.command = GEOMETRY;
        geomPacket.geomCylinders = LocalMaxCylinders;
        geomPacket.geomHeads = LocalMaxHeads;
        geomPacket.geomSectors = LocalMaxSectors;
        geomPacket.firstCyl = ImageFileHeader.partitionStartCylinder;
        geomPacket.lastCyl = ImageFileHeader.partitionEndCylinder;
        /* Fill in the IPX packet header */
        /* NETWORK number: */
        memmove( header.destination.network, NetAddress, 4);
        /* NODE number: BROADCAST */
        memset( header.destination.node, 0xff, 6);
```

```
        /* SOCKET number: Our special number. */
        *(WORD *)header.destination.socket = SWAP_WORD( DOWNLOAD_SOCKET_NUMBER);
        header.packetType = 4;
        /* Fill in the ECB. */
        sendECB.ESRAddress = NULL;
        sendECB.socketNumber = LocalSocket;
        sendECB.fragmentCount = 2;
        sendECB.fragmentDescriptor[0].address = &header;
        sendECB.fragmentDescriptor[0].size = sizeof( header);
        sendECB.fragmentDescriptor[1].address = &geomPacket;
        sendECB.fragmentDescriptor[1].size = sizeof( geomPacket);
        memset( sendECB.immediateAddress, 0xff, 6);        /* Broadcast. */
        /* Send the packet. */
        IPXSendPacket( &sendECB);
        while ( sendECB.inUseFlag) {
                IPXRelinquishControl();
        }
        return sendECB.completionCode;
}
/*********************************************************************/
/* Send an EXIT broadcast packet.
 *
 * Return ZERO if successful, nonZero otherwise.
 */
int
sendFarewellBroadcastPacket( BYTE command)
{
        ECB     sendECB;
        IPXHeader       header;
```

```
    FarewellPacket    packet;
    /* Fill in the packet data
     */
    packet.command = command;
    /* Fill in the IPX packet header */
    /* NETWORK number: */
    memmove( header.destination.network, NetAddress, 4);
    /* NODE number: BROADCAST */
    memset( header.destination.node, 0xff, 6);
    /* SOCKET number: Our special number. */
    *(WORD *)header.destination.socket = SWAP_WORD( DOWNLOAD_SOCKET_NUMBER);
    header.packetType = 4;
    /* Fill in the ECB. */
    sendECB.ESRAddress = NULL;
    sendECB.socketNumber = LocalSocket;
    sendECB.fragmentCount = 2;
    sendECB.fragmentDescriptor[0].address = &header;
    sendECB.fragmentDescriptor[0].size = sizeof( header);
    sendECB.fragmentDescriptor[1].address = &packet;
    sendECB.fragmentDescriptor[1].size = sizeof( packet);
    memset( sendECB.immediateAddress, 0xff, 6);        /* Broadcast. */
    /* Send the packet. */
    IPXSendPacket( &sendECB);
    while ( sendECB.inUseFlag) {
        IPXRelinquishControl();
    }
    return sendECB.completionCode;
}
/*************************************************************************
```

```
        Program:  BroadcastDriveGeometry

        Description: This function sends the drive geometry to the image
                slaves on the send IPX socket.

        Author: Kevin J. Turpin        Date: 14 May  1996
*************************************************************************/
void BroadcastDriveGeometry( void)
{
        clock_t  nextSendTime = 0;
        ECB far *ecb;
        /* Initialize the "currently connected clients" display. */
        DisplayClients();
        while ( ! kbhit()) {
                /*******************************************/
                IPXRelinquishControl();   /* Allow packet processing to occur. */
                /*******************************************/
                /* Periodically send geometry packets. */
                if ( clock() > nextSendTime) {
              nextSendTime = clock() + CLK_TCK/2;     /* Schedule next packet. */
                if ( sendGeometryBroadcastPacket()) {
                        sprintf(ErrorMessage, "\n\nError sending geometry
                                broadcast packet");
                        LogError(SENDPACKETERR, ErrorMessage);
                        CleanExit();
                        }
                        }
                        /*******************************************/
                        /* If we RECEIVE any packets, process them. */
                        while ( ( ecb = receivedPacket()) != NULL) {
```

```
                if ( ecb->completionCode != 0) {
                        DB_INFORM( "Receive packet error: ccode 0x%x\n", ecb-
                                >completionCode);
                        }
                        else {
                        /* Process it only if it's an RSVP packet.
                         * First byte of data is the "command".
                         */
                        BYTE command = *(BYTE *)ecb->fragmentDescriptor[ 1].address;
                if ( command == RSVP) {
                if ( addClientToList( ecb)) {
                DB_INFORM( "Error adding client to list.\n");
        }
        else {
                DB_INFORM( "Added client.\n");
                if ( sendRSVPAck( ecb)) {
                        DB_INFORM( "Error sending RSVP ACK packet.\n");
                }
                DB_INFORM( "Sent ACK.");
        }
    }
    }
    IPXListenForPacket( ecb);/* Repost the listen. */
    }
    /* If the current client count exceeds our licensed client count,
     * print a big bad error message and exit.
     */
    if ( currentClientCount > licenses) {
            sprintf(ErrorMessage, "\n\nToo many clients for current license");
```

```
            LogError(LICENSEERR, ErrorMessage);
            CleanExit();
            }
    /* If all clients are with us, we can go ahead and get started. */
    if (  expectedClientCount != -1
            && expectedClientCount == currentClientCount) {
            /* All clients ready, so let's get started with the download. */
            break;
            }
    }
    DB_INFORM( "\nDONE Sending drive geometry information.\n");
    /* Gobble keystrokes, if any... */
    while ( kbhit()) {
            getch();
    }
    }
    /*************************************************************************
            Program:  SendOurIPXPacket
            Description: This function sends the IPX packet on the IPX send
                    socket.
            Author: Kevin J. Turpin         Date: 14 May  1996
    *************************************************************************/
    /* Send an ipx packet with default size. */
    void SendOurIPXPacket( void)
    {
            SendOurIPXPacketSized( 512 + sizeof(OurHeader));
    }
    /*****************************************/
    void
```

```
SendOurIPXPacketSized( int dataSize)
{
        SendHeader.packetType = 4;
        // Stay on our segment
        memmove(SendHeader.destination.network, NetAddress, 4);
        // But let any node on our segment get it (broadcast)
        for (i=0; i<6; i++) {
        SendHeader.destination.node[i] = 0xff; /* node address of slave*/
        }
        *(WORD *) (SendHeader.destination.socket) = SWAP_WORD(
                DOWNLOAD_SOCKET_NUMBER);
        /* Setup ECBs */
        SendECB.ESRAddress = 0;
        SendECB.socketNumber = LocalSocket;
        SendECB.fragmentCount = 2;
        SendECB.fragmentDescriptor[0].address = &SendHeader;
        SendECB.fragmentDescriptor[0].size = 30;
        SendECB.fragmentDescriptor[1].address = IPXSendBuffer;
        SendECB.fragmentDescriptor[1].size = dataSize;
        for (i=0; i<6; i++) {
                SendECB.immediateAddress[i] = 0xFF;/* broadcast packet */
        }
        memmove(IPXSendBuffer, &OurHeader, sizeof(OurHeader));
        IPXSendPacket(&SendECB);
        while (SendECB.inUseFlag !=0)
                IPXRelinquishControl();
        /* If error sending packet, let calling function know */
        if (SendECB.completionCode != 0)
                printf("\nError sending packet!");
```

```
}
/*************************************************************************
        Program:  ReadDriveHeadAt
        Description:   Generic bios disk interface.
                If there are any errors, they are reported.  The process
                is aborted after 5 retries on errors.
                CRC type errors are not necessarily fatal, so we will not
                abort on them (code = 0x11).
*************************************************************************/
void ReadDriveHeadAt( int drive,
                        int cylinder,
                        int head,
                        int sector,
                        int sectorCount,
                        unsigned char * buffer)
{
        int ccode,
                retryCount = 0;
        do {
                // Try to read the head
                ccode = biosdisk(READ,
                        drive,
                        head,
                        cylinder,
                        sector,
                        sectorCount,
                        buffer);
                if ( ccode) {
                        DisplayBiosError( ccode, 20, 25);
```

```
                }
        } while ( ccode && retryCount++ < 5);
        if ( ccode && ccode != 0x11) {      /* Tolerate CRC "errors". */
                sprintf(ErrorMessage,
                        "\n\nBios error %d reading at cylinder %d, head %d",
                        ccode,
                        cylinder,
                        head);
                LogError(BIOSERR, ErrorMessage);
                CleanExit();
        }
}
/*************************************************************************
        Program:  ReadDriveHead
        Description: This function reads a head worth of data from the
                local     hard drive at the global variable CurrentCylinder and
                CurrentHead.
                If there are any errors, they are reported.  The process
                is aborted after 5 retries on errors.
                CRC type errors are not necessarily fatal, so we will not
                abort on them (code = 0x11).
        Author: Kevin J. Turpin        Date: 14 May  1996
*************************************************************************/
void ReadDriveHead(unsigned char * buffer)
{
        ReadDriveHeadAt( CurrentHDDrive,
                        CurrentCylinder,
                        CurrentHead,
                        1,
```

```
                        LocalMaxSectors,
                        buffer);
    }
    /*********************************************************************/
    /* Send data for an entire HEAD.
     */
    void
    BroadcastGivenHead( long cylinder,
                        long head,
                        unsigned char *buffer,
                        long numSectors)
    {
        long i;
    int dropIt = 0;
    // Stuff the header for this head
    OurHeader.command = STD_DATA;
    OurHeader.head = head;
    OurHeader.cylinder = cylinder;
    // Send each sector of data, packet it and send it
    for (i=0; i<numSectors; i++){
        memmove(&IPXSendBuffer[sizeof(OurHeader)],
            &buffer[ (int)i*512], 512);
        OurHeader.sector = i+1;
        if (i == numSectors-1)      // on the last sector, write it
        OurHeader.command = FLUSH;
        tenthMilliDelay(DelayValue);      /* Inter-packet delay. */
    /*********************************************/
    if ( DebugBroadcast) {
        /* FAKE DROPPING A PACKET.
```

```
* also allow keyboard control of delay values
*/
if ( kbhit()) {
        switch( getch()) {
        case '*':
                if ( FlushDelay > 0) {
                        FlushDelay--;
                }
                showDelayValues();
                break;
        case '9':
                FlushDelay++;
                showDelayValues();
                break;
        case '-':
                if ( DelayValue > 0) {
                        DelayValue--;
                        }
                showDelayValues();
                break;
        case '+':
                DelayValue++;
                showDelayValues();
                break;
        case '/': { /* Show the perDelay values. */
                int d;
                int x=wherex(), y=wherey();
                for ( d = 0 ; d < 20 ; d++) {
                gotoxy(40,d+1);
```

```
                        cprintf("d %d: m %ld, s %ld",
                                d,
                                perDelayCounts[ d].headsMissed,
                                perDelayCounts[ d].headsSent);
                        /* Avoid divide-by-zero errors */
                        if ( perDelayCounts[ d].headsSent != 0) {
                        cprintf(", ratio %5.4f",
                                ( (float)perDelayCounts[ d].headsMissed
                                / (float)perDelayCounts[ d].headsSent));
                        }
                        clreol();
                    }
                    gotoxy( x, y);
                    break;
            }
            default:
                    DB_INFORM( "DROPPED A PACKET...\n");
                    dropIt = 1;
                    break;
            }
    }
                        /*************************/
            if ( dropIt) {
                    dropIt = 0;
                    continue;
                    }
            }
    #if defined( KEY_ENABLE_DEBUG)
            else {
```

```
                if ( kbhit()) {
                        if ( getch() == '@') {
                                if ( DebugBroadcast != TRUE) {
                                        DebugBroadcast = TRUE;
                                        DB_INFORM( "DebugBroadcast Enabled.\n");
                                }
                                else {
                                        DB_INFORM( "DebugBroadcast Disabled.\n");
                                        DebugBroadcast = FALSE;
                                }
                        }
                }
        }
#endif
                /*********************************************/
                SendOurIPXPacket();
        }
        tenthMilliDelay(FlushDelay);
}
/*****************************************************************/
/* Send a "skip this track" command.
 */
void
BroadcastSkipHead( long cylinder,
                        long head)
{
        ECB     sendECB;
        IPXHeader header;
        SkipTrackPacket stp;
```

```
/* Fill in the packet data
 */
stp.command = SKIP_TRACK;
stp.cylinder = cylinder;
stp.head = head;
/* Fill in the IPX packet header */
/* NETWORK number: */
memmove( header.destination.network, NetAddress, 4);
/* NODE number: BROADCAST */
memset( header.destination.node, 0xff, 6);
/* SOCKET number: Our special number. */
*(WORD *)header.destination.socket = SWAP_WORD(
        DOWNLOAD_SOCKET_NUMBER);
header.packetType = 4;
/* Fill in the ECB. */
sendECB.ESRAddress = NULL;
sendECB.socketNumber = LocalSocket;
sendECB.fragmentCount = 2;
sendECB.fragmentDescriptor[0].address = &header;
sendECB.fragmentDescriptor[0].size = sizeof( header);
sendECB.fragmentDescriptor[1].address = &stp;
sendECB.fragmentDescriptor[1].size = sizeof( stp);
memset( sendECB.immediateAddress, 0xff, 6);      /* Broadcast. */
tenthMilliDelay(DelayValue);      /* Inter-packet delay. */
/*****************************************/
/* Send the packet. */
IPXSendPacket( &sendECB);
while ( sendECB.inUseFlag) {
        IPXRelinquishControl();
```

```
        }
        if ( sendECB.completionCode) {
                DB_INFORM( "Error sending SKIP packet.\n");
        }
}
/*************************************************************************/
/* Retransmit a track--it was missed by one or more clients.
 */
void
retransmitTrack( long cylinder,
                        long head)
{
        /* Check to make sure the track is valid on the disk.
         * We make sure the requested track isn't beyond the "valid
         * up thru" track that we know is on the disk.
         */
        if ( ! (  CHlessThan( cylinder, head,
        ValidUpThruCylinder, ValidUpThruHead)
                || CHEqual( cylinder, head,
        ValidUpThruCylinder, ValidUpThruHead))) {
DB_INFORM( "Ignoring resend request: (%ld,%ld) GT validUpThru (%ld,%ld).\n", cylinder,
head,(long)CurrentCylinder, (long)CurrentHead);
                return;
        }
/* #define NO_INLINE_REBROADCASTS */
#if defined( NO_INLINE_REBROADCASTS)
        if ( Phase == BLAST) {
                DB_INFORM( "Client(s) Lost Track (%ld,%ld).\n", cylinder, head);
                return;
```

```
        }
#endif
        /* Read the track from the local disk into the HeadBuffer */
        ReadDriveHeadAt( CurrentHDDrive,
                        (int)cylinder,
                        (int)head,
                        1,
                        LocalMaxSectors,
                        HeadBuffer);
        /* Now send the entire track. */
        BroadcastGivenHead( cylinder,
                            head,
                            HeadBuffer,
                            LocalMaxSectors);
        DB_INFORM( "Resent track (%ld,%ld).\n",
                    cylinder, head);
}
/*************************************************************************/
/* Return TRUE if we really slowed down. */
int
slowDown( void)
{
        DelayValue++;
        DB_INFORM( "Slowed to delay %d", DelayValue);
        showDelayValues();
        return 1;
}
/************************/
/* Return TRUE if we really sped up. */
```

```
int
speedUp( void)
{
        int i;
        int lowestNonZeroRatioIndex = -1;
        float   current_ratio;
        if( Phase != BLAST) {
                return 0;
        }
        if ( DelayValue > 0) {
                if ( DelayValue >= NUM_DELAYS) {
                        DB_INFORM( "Speeding up incrementally: Delay is
                                BIG.\n");
                        DelayValue--;
                        showDelayValues();
                        return 1;
                }
                /*********************************************/
                /* Find the delay slot with the best "eligible" rating.
                 * A delay value is eligible if we've sent less than
                 * 50 heads at that value, or if we've sent more than
                 * 50 heads and missed fewer than two percent of them,
                 * but NEVER if the consecutiveDataLosses flag is set
                 * for that speed.
                 */
                for ( i = 0 ; i < NUM_DELAYS ; i++) {
                        if ( perDelayCounts[ i].consecutiveDataLosses) {
                                continue;       /* This one's NOT eligible. */
                        }
```

```
                        if ( perDelayCounts[ i].headsSent < 50) {
                                /* Hey, this is the one we want! */
                                lowestNonZeroRatioIndex = i;
                                break;
                        }
        current_ratio = (  (float)perDelayCounts[ i].headsMissed
/ (float)perDelayCounts[ i].headsSent);
        if ( current_ratio <= 0.02) {
                /* We want this one. */
                lowestNonZeroRatioIndex = i;
                break;
                }
        }
        if ( lowestNonZeroRatioIndex == -1) {
                /* NO desirable ratio. */
                DB_INFORM( "NOT speeding up: no desirable ratio.\n");
                showDelayValues();
                return 0;
        }
        if ( DelayValue > lowestNonZeroRatioIndex) {
                DB_INFORM( "Speeding up: current %u > want %u\n",
                        DelayValue, lowestNonZeroRatioIndex);
                        DelayValue = lowestNonZeroRatioIndex;
        /* Jump right to our goal. */
                        showDelayValues();
                        return 1;
                }
        DB_INFORM( "NOT speeding up: current %u <= want %u\n",
                DelayValue, lowestNonZeroRatioIndex);
```

```
        }
        else {
        DB_INFORM( "NOT speeding up: Delay already Zero.\n");
        }
        showDelayValues();
        return 0;
}
/*************************************************************************/
typedef struct {
        long cylinder;
        long head;
        unsigned int count;          /* Count of requests for this track. */
} ResendRequest;
ResendRequest resendRequests[ NUMBER_ECBs];
int                                          totalResendRequests = 0;
/*****************************************/
/* We shouldn't have more unique entries here than we have ECBs. This  * is because we clear and recreate
the list approximately once per "ecb * listen post" loop.
*/
void
zeroResendRequests( void)
{
        int i;
        for ( i = 0 ; i < NUMBER_ECBs ; i++) {
                resendRequests[ i].cylinder = resendRequests[ i].head = -1L;
                resendRequests[ i].count = -1;
        }
        totalResendRequests = 0;
}
```

```
/*********************************************/
/* A client has requested a resend of a track.  Record the cylinder and     * head, and bump the count of
clients that have requested that track.
 */
void
recordResendRequest( long cylinder,
                     long head)
{
        int i;
        for ( i = 0 ; i < NUMBER_ECBs ; i++) {
                if (  resendRequests[ i].cylinder == cylinder
                        && resendRequests[ i].head == head) {
                /* Found slot. Just bump the count and we're done. */
                resendRequests[ i].count++;
                totalResendRequests++;
                return;
        }
        /* Else the slot didn't match.  If the slot is empty, record the
         * new track request and we're done.
         */
        if ( resendRequests[ i].cylinder == -1L) {
                resendRequests[ i].cylinder = cylinder;
                resendRequests[ i].head = head;
                resendRequests[ i].count = 1;
                totalResendRequests++;
                return;
        }
}
/* Zowee! Out of slots in the resend table! */
```

```
DB_INFORM( "Out of slots in resend table.  Dropping resend request.\n");
}
/*********************************************/
/* Process the resendRequests list.  Send the tracks and update the
 * delay counters if we have enough clients requesting a particular *  * track.
 *
 * Return number of heads missed.
 *
 */
int
processResendList( void)
{
        int i;
        unsigned int missedHeadCount = 0;
        unsigned int missedHeadSum;
        unsigned int nConsecutiveLostData = 0;
        unsigned int nMaxConsecutiveLostData = 0;
        unsigned int nClientRequests = 0;
        /* First see if we need to slow down. */
        for ( i = 0 ; i < NUMBER_ECBs ; i++) {
                if ( resendRequests[ i].cylinder == -1L) {
                        /* No more requests. */
                        break;
                }
                /* Sum up total of client requests. */
                nClientRequests += resendRequests[ i].count;
                missedHeadCount++;      /* Bump local counter. */
                totalHeadsMissed++;     /* Update global counter. */
        }
```

```
        /*****************************************/
    /* If we miss some data,put off the next speedup attempt for a while. */
        if ( missedHeadCount) {
                timeForNextSpeedup = NEXT_SPEEDUP();
        }
        /*****************************************/
        /* Compute sliding window of lost data--used to
         * trigger slowdowns.
         */
        numberSamplesThisWindow++;
        windowSamples[ windowSlot] = nClientRequests;
        for ( i = 0, missedHeadSum = 0 ; i < WINDOW_SIZE ; i++) {
                /* UPdate counter of total slots with missed heads. */
                if ( windowSamples[ i]) {
                        missedHeadSum++;
                }
        /* Update maximum run of slots with more than (say) ten percent
         * of clients reporting
         */
                if ( windowSamples[ i] > ( currentClientCount / 10)) {
                        nConsecutiveLostData++;
                        if ( nConsecutiveLostData > nMaxConsecutiveLostData) {
                                nMaxConsecutiveLostData = nConsecutiveLostData;
                        }
                }
                else {
                        nConsecutiveLostData = 0;
                }
        }
```

```
/*********************************************/
/* Slow down if we have more than '2n' consecutive missed heads,
 * OR if we have more than 'n' missed heads in the window.
 */
if (  ( nMaxConsecutiveLostData > ( 2 * SLOWDOWN_THRESHOLD))
        ||(  numberSamplesThisWindow > WINDOW_SIZE
                && missedHeadSum > SLOWDOWN_THRESHOLD)) {
/* If we're slowing down due to consecutive data losses,
 * mark this delay value ineligible for speedup consideration.
 */
    if ( nMaxConsecutiveLostData > ( 2 * SLOWDOWN_THRESHOLD)) {
        if ( 0 <= DelayValue && DelayValue < NUM_DELAYS) {
        perDelayCounts[DelayValue].consecutiveDataLosses = 1;
        }
        else {
            outOfRangeCounts.consecutiveDataLosses = 1;
        }
    }
    slowDown();     /* Bump the delay. */
/* Reset window--start collecting data on lost heads from scratch. */
    for ( i = 0 ; i < WINDOW_SIZE ; i++) {
        windowSamples[ i] = 0;
    }
    windowSlot = 0;
    numberSamplesThisWindow = 0L;
        /* Reset the next speedup time so we don't
         * speed right back up again.
         */
        timeForNextSpeedup = NEXT_SPEEDUP();
```

```
            /* Clear the perDelay values for the slower speed so that
             * we start acquiring new data.
             */
            if ( 0 <= DelayValue && DelayValue < NUM_DELAYS) {
                perDelayCounts[ DelayValue].headsSent = 0;
                perDelayCounts[ DelayValue].headsMissed = 0;
            }
        }
        /*********************************************/
        /* Resend the track(s) at the (possibly slower) speed. */
        for ( i = 0 ; i < NUMBER_ECBs ; i++) {
            if ( resendRequests[ i].cylinder == -1L) {
                /* No more requests. */
                break;
            }
            retransmitTrack( resendRequests[ i].cylinder,
                    resendRequests[ i].head);
        }
        /*********************************************/
        /* Update the perDelay values. */
        if ( Phase == BLAST) {
            if ( DelayValue < NUM_DELAYS) {
                perDelayCounts[ DelayValue].headsSent++;
                if ( missedHeadCount) {
                    perDelayCounts[ DelayValue].headsMissed++;
                }
            }
            else {
                outOfRangeCounts.headsSent++;
```

```
                        if ( missedHeadCount) {
                                outOfRangeCounts.headsMissed++;
                        }
                }
        }
        /*******************************************/
        /* Speed up if we've had a number of good tracks in a row.
         */
        if ( clock() > timeForNextSpeedup) {
                speedUp();
                timeForNextSpeedup = NEXT_SPEEDUP();
        }
        /*******************************************/
        /* Update the "lost data" window. */
        windowSlot = ( windowSlot + 1) % WINDOW_SIZE;
        return missedHeadCount;
}
/*************************************************************************/
/* Clients are leaving us.  Remove them from our client list and
 * send them a "farewell ACK" packet.
 */
BYTE
processFarewellPacket( IPXHeader *reqHdr)
{
        ClientInfo *c;
        ECB     sendECB;
        IPXHeader header;
        FarewellPacket  packet;
        int transportTime;
```

```
for ( c = clients ; c != NULL ; c = c->next) {
        if ( memcmp( c->netAddress, &reqHdr->source, 12) == 0) {
                /* Found the client: Remove it from our list. */
                if ( c->prev) {
                        c->prev->next = c->next;
                }
                if ( c->next) {
                        c->next->prev = c->prev;
                }
                if ( c == clients) {
                        clients = c->next;
                }
                currentClientCount--;/* One less client on list. */
                /* Free the storage. */
                free( c);
                break;/* DON'T continue loop: variable 'c' nowinvalid,
                * and we're done anyway because the client has been
                 * removed from the list.
                 */
        }
}
/* Send back a farewell ACK packet. */
/* Fill in the RSVP packet data
 */
packet.command = FAREWELL_ACK;
/* Fill in the IPX packet header */
memmove( &header.destination,
        &reqHdr->source,
        sizeof( reqHdr->source));
```

```
        header.packetType = 4;
        /* Fill in the ECB. */
        sendECB.ESRAddress = 0;
        sendECB.socketNumber = LocalSocket;
        sendECB.fragmentCount = 2;
        sendECB.fragmentDescriptor[0].address = &header;
        sendECB.fragmentDescriptor[0].size = sizeof( header);
        sendECB.fragmentDescriptor[1].address = &packet;
        sendECB.fragmentDescriptor[1].size = sizeof( packet);
        if ( IPXGetLocalTarget( (BYTE far *)&reqHdr->source,
                                        sendECB.immediateAddress,
                                        &transportTime)) {
                return 0xff;        /* Error. */
        }
        /* Send the packet. */
        IPXSendPacket( &sendECB);
        while ( sendECB.inUseFlag) {
                IPXRelinquishControl();
        }
        return sendECB.completionCode;
}
/*********************************************************************/
/* Process packets from the clients.
 *
 * Return count of packets that were processed.
 *
 */
long
processIncomingPackets( void)
```

```
{
        long processedPackets = 0L;
        ECB far *ecb;
        do {
                /* Clear the list of resend requests. */
                zeroResendRequests();
                while ( ( ecb = receivedPacket()) != NULL) {
                        processedPackets++;
                        if ( ecb->completionCode != 0) {
DB_INFORM( "Receive packet error: ccode 0x%x\n", ecb->completionCode);
                        }
                        else {
                                /* Process only "resend" and "farewell" packets.
                                 * First by of data is the "command".
                                 */
                        BYTE command = *(BYTE *)ecb->fragmentDescriptor[ 1].address;
                                switch( command) {
                                case RESEND_REQUEST: {
    ResendPacket *rp = (ResendPacket *)ecb->fragmentDescriptor[ 1].address;
            recordResendRequest( rp->cylinder, rp->head);
    /* If in big debug mode, display MAC addresses of clients that missed
     * packets.
     */
            if ( DebugBroadcastDisplayMACs) {
            IPXHeader *hdr = (IPXHeader *)ecb->fragmentDescriptor[ 0].address;
            DB_INFORM( "Client requested resend. MacAddr: 0x%02x%02x%02x%02x%02x%02x%02x\n",
                    hdr->source.node[ 0], hdr->source.node[ 1],
                    hdr->source.node[ 2], hdr->source.node[ 3],
                    hdr->source.node[ 4], hdr->source.node[ 5]);
```

```
                }
                else {
                        DB_INFORM( "RecordedResentRequest.\n");
                }
                break;
        }
        case FAREWELL: {
        IPXHeader *hdr = (IPXHeader *)ecb->fragmentDescriptor[ 0].address;
        BYTE ccode;
        ccode = processFarewellPacket( hdr);
        if ( ccode) {
        DB_INFORM( "Error 0x%x from processFarewellPacket.\n", ccode);
        }
        else {
                DB_INFORM( "Successful processFarewellPacket().\n", ccode);
        }
        break;
}
        default:
        DB_INFORM( "pIP: Ignoring unknown packet cmd 0x%x.\n", command);
                break;    /* Ignore other packets. */
        }
}
IPXListenForPacket( ecb);/* Repost the listen. */
}
/* Now send the missing tracks.  If we resent any,
 * run the loop again to check for more lost data.
 */
} while ( processResendList());
```

```
return processedPackets;
}
/*****************************************************************************/
/*
 * Send "EXIT" (AKA no more data) packets every second or so.
 * Respond to RESEND_REQUEST packets with track resends.
 * If we don't get any RESEND_REQUESTS for a while,
 * we're done.
 */
void
handleLateResendRequests( void)
{
        clock_t  nextSendTime = 0;
        clock_t  startOfIdle = clock();
        while ( 1) {
                /*********************************************/
                IPXRelinquishControl();  /* Allowpacketprocessingto occur. */
                /*********************************************/
                /* Periodically send EXIT packets. */
                if ( clock() > nextSendTime) {
                nextSendTime = clock() + CLK_TCK/2;     /*Schedule next packet*/
                        if ( sendFarewellBroadcastPacket( EXIT)) {
        sprintf(ErrorMessage, "\n\nError sending EXIT broadcast packet");
        LogError(SENDPACKETERR, ErrorMessage);
        CleanExit();
        }
}
if( processIncomingPackets()) {
        /* Packet was processed, so restart the idle timer. */
```

```
            startOfIdle = clock();
            /* Also send another EXIT packet to prompt remaining clients
             * to send us RESEND requests NOW.
             */
            nextSendTime = clock() - 1;
        }
        /* BAIL OUT if timeout. */
        if ( ( clock() - startOfIdle) > ( 5 * CLK_TCK)) {
                DB_INFORM( "Leaving EXIT phase due to idle timeout.\n");
                break;   /* TIMEOUT */
        }
                /* BAIL OUT if key pressed. */
    #if defined( KEY_ENABLE_DEBUG)
                if ( kbhit()) {
                        while ( kbhit()) {
                                if ( getch() == '@') {
                                        if ( DebugBroadcast != TRUE) {
                                                DebugBroadcast = TRUE;
                                        DB_INFORM( "DebugBroadcast Enabled.\n");
                                        }
                                        else {
                                        DB_INFORM( "DebugBroadcast Disabled.\n");
                                        DebugBroadcast = FALSE;
                                        }
                                }
                                else {
                                DB_INFORM( "Leaving EXIT phase due to keypress.\n");
                                return;
                                }
```

```
                }
            }
#else
        if ( kbhit()) {
            while ( kbhit()) {
                (void)getch();
                }
                DB_INFORM( "Leaving EXIT phase due to keypress.\n");
                break;
            }
#endif
        }
}
/*************************************************************************/
/* Print a list of clients that didn't disconnect (yet).
 */
void
showHangingClients( void)
{
        ClientInfo *c;
        int i;
        inform( stderr, "\nThe following clients have not yet disconnected:\n");
        for ( c = clients, i = 0 ; c != NULL ; c = c->next) {
            inform( stderr,
                "%02x%02x%02x%02x%02x%02x ",
                c->netAddress[ 4],
                c->netAddress[ 5],
                c->netAddress[ 6],
                c->netAddress[ 7],
```

```
                        c->netAddress[ 8],
                        c->netAddress[ 9]);
                if ( ++i >= 4) {
                        i = 0;
                        inform( stderr, "\n");
                }
        }
}
/*****************************************************************************/
/*
 * Send "GOODBYE" packets every second or so.
 * Respond to "FAREWELL" packets by removing the client
 * from our list and sending back "FAREWELL_ACK" packets.
 * When we get get no requests for a while, OR our client
 * list becomes empty, we're done.
 */
void
sayGoodbyeToClients( void)
{
        clock_t  nextSendTime = 0;
        clock_t  startOfIdle = clock();
        while ( 1) {
                /*******************************************/
                IPXRelinquishControl();  /* Allow packet processingto occur*/
                /*******************************************/
                /* Periodically send GOODBYE packets. */
                if ( clock() > nextSendTime) {
                        nextSendTime = clock() + CLK_TCK/2;
        /* Schedule next packet. */
```

```
                    if ( sendFarewellBroadcastPacket( GOODBYE)) {
        sprintf(ErrorMessage, "\n\nError sending DISCONNECT broadcast packet");
                LogError(SENDPACKETERR, ErrorMessage);
                    CleanExit();
                    }
                }
                if( processIncomingPackets()) {
                /* Packet was processed, so restart the idle timer. */
                startOfIdle = clock();
                }
                /* We're DONE if the client list becomes empty. */
                    if ( clients == NULL) {
                        TopLargeMsg( 0, "  DONE   ");
        inform( stderr, "\n\nAll clients have disconnected.  We're done.\n");
                        break;
                    }
                    /* BAIL OUT if timeout. */
                    if ( ( clock() - startOfIdle) > ( 5 * CLK_TCK)) {
        inform( stderr, "\n\nLeaving DISCONNECT phase due to idle timeout.\n");
                showHangingClients();
                break;   /* TIMEOUT */
                }
                /* BAIL OUT if key pressed. */
        #if defined( KEY_ENABLE_DEBUG)
                if ( kbhit()) {
                    while ( kbhit()) {
                        if ( getch() == '@') {
                            if ( DebugBroadcast != TRUE) {
                                DebugBroadcast = TRUE;
```

```
                                        DB_INFORM( "DebugBroadcast Enabled.\n");
                                        }
                                        else {
                                        DB_INFORM( "DebugBroadcast Disabled.\n");
                                        DebugBroadcast = FALSE;
                                        }
                                }
                                else {
        inform( stderr, "\n\nLeaving DISCONNECT phase due to keypress.\n");
                                showHangingClients();
                                return;
                                }
                        }
                        }
        #else
                        if ( kbhit()) {
                        while ( kbhit()) {
                                (void)getch();
                        }
        inform( stderr, "\n\nLeaving DISCONNECT phase due to keypress.\n");
                        showHangingClients();
                        break;
                        }
        #endif
        }
        }
        /*************************************************************************/
        /* Finish broadcast processing.
         */
```

```
void
finishBroadcastProcessing( void)
{
        Phase = CLEANUP;
        TopLargeMsg( 0, " FINISHING ");
        handleLateResendRequests();
        TopLargeMsg( 0, "DISCONNECT ");
        sayGoodbyeToClients();
}
/*************************************************************************
        Program:  BroadcastHead
        Description:   This function broadcasts the drive head of data (HeadBuffer)to the image slaves
        using the send IPX socket. The data is sent 512 bytes at a time (sector of data). On the last
        sector, the command to flush the data (write it to the hard drive) is sent.
        Author: Kevin J. Turpin        Date: 14 May  1996
*************************************************************************/
void BroadcastHead( void)
{
/* Typically, we only broadcast a head when it contains valid data.
 * But the ForceWriteData flag can override and cause us to always
 * write the real data.
 */
if ( ValidDataInHead == TRUE || ForceWriteData == TRUE) {
        BroadcastGivenHead( CurrentCylinder,
                            CurrentHead,
                            HeadBuffer,
                            LocalMaxSectors);
}
        else {
```

```
                BroadcastSkipHead( CurrentCylinder,
                                   CurrentHead);
        }
        /*****************************************/
        /* Process any incoming packets: If we have a "resend"
         * request, read that head from the disk and send it.
         */
        processIncomingPackets();
        totalHeadsSent++;          /* Update and periodically display counters. */
        displayTotalAndMissedHeads();
    }
/*************************************************************************
        Program: CompressHead
        Description:  This function is the main compression controller for this program.  It is responsible
        for taking the HeadBuffer and
        compressing the data, writing it to CompressBuffer, and
        flushing the CompressBuffer when it is full (or close to
        full).
        The "Run-Length" compression algorithm used in the old
        PCX graphics files will be used.  This compression algorithm
        compresses by counting the number of consecutive bytes
        that contain the same data and then representing this
        data as three bytes:
                BYTE            DATA
                ----            ----
                1               Compression Key (ID)
                2               Compression Count (Run Count)
                3               Data
        The first byte indicates that the next two bytes contain
```

```
        compressed data.  The second byte represents the number of
        times the data (third byte) is replicated.
        The compression Key (ID) should be somewhat unique (not
        highly probable to be on the drive).  Novell's disk image
        program uses a key id of 0xD5.  We will use the same key
        to make this program capable of uncompressing Novell's
        images in the event Novell uses our lab and wants to use
        their own images.
        There may be multiple HeadBuffers compressed into a single
        CompressedBuffer.  Therefore, we will keep some pointers
        between HeadBuffers.  The following pointers/counters are
        used for the compression:
                LastData - contains the last byte of data being compressed
                Current - contains the current byte being evaluated
                CompressCount - counts the number of bytes that are the

                same.
        Returns ZERO on success, nonZero on failure.
        Author: Kevin J. Turpin         Date: 15 May  1996
**************************************************************************/
int
CompressHead( void)
{
        int i;
        int error = 0;/* Always successful, unless
                        * WriteDataToCompressBuffer fails.
                        */
        // Let's start at the begining of the HeadBuffer
        LastData = HeadBuffer[0];
```

```
// We start clean on each head.
CompressCount = 1;
// Now run through the HeadBuffer
for (i=1; i<LocalMaxSectors*512; i++) {
        Current = HeadBuffer[i];
        // Temporary test (Kevin)
        if (Current != FILLDATA &&
                ValidDataInHead != TRUE)
                ValidDataInHead = TRUE;
        // We only need to work with the compress buffer if we are
        // writing to an image file.
        if (ImageFileNameValid == TRUE) {
// If current is the same as last, it can be compressed, count it
        if (Current == LastData) {
                CompressCount++;
                if (CompressCount == 255) {// largest count in a byte
                        error |= WriteDataToCompressBuffer();
                        CompressCount = 0;//reset to 0since we are still
                        }                       // compressing.
                }
                else {          // does not compress any longer
                if (CompressCount > 0) {//If we were compressingbefore
                        error |= WriteDataToCompressBuffer();
                // move it to the CompressBuffer
                }
                LastData=Current;       // get ready for next evaluation
                CompressCount = 1;// reset to 1 since the next byte
        }                               // may not compress.
}
```

```
    }

        // for simplicity, we will not try to compress across HeadBuffers
        // therefore, if we have data that needs to be written to the
        // CompressBuffer, let's do it.
        if (CompressCount > 0 &&
                ImageFileNameValid == TRUE) {
            error |= WriteDataToCompressBuffer();
            CompressCount = 1;
    }
        return error;
    }
    /*************************************************************************
        Program:  WriteDataToCompressBuffer
        Description:   This function writes the compress data into the
        CompressBuffer.  Compressed data is represented in
        three bytes of data (as explained in CompressHead function).
        Therefore, it the CompressCount is less than 3, there is
        no savings in representing the data in compressed format.
        When the CompressBuffer is almost full (within 3 bytes of
        its limit), we will flush it to the image file.
        Returns ZERO on success, nonZero on failure.
        Author: Kevin J. Turpin        Date: 6 May  1996
    Mod 1: Added display of counters for debug purposes.Kevin - Oct 29, 1996
    *************************************************************************/
    int
    WriteDataToCompressBuffer( void)
    {
        int error = 0;
            /* Always successful, unless FlushCompressBuffer fails. */
```

```
// Is it efficient to represent the data in compressed format?
// If data is same as compression key, put it in compressed format so
// that it doesn't confuse us during uncompressing.
        if ((CompressCount > 2) ||
                (LastData == COMPRESSIONKEY)) {      // yes
        CompressBuffer[(unsigned int)CompressPTR++] = COMPRESSIONKEY;
        CompressBuffer[(unsigned int)CompressPTR++] = CompressCount;
CompressBuffer[(unsigned int)CompressPTR++] = LastData;//actual data
        // If we need debug info, display Compressed Key count value
        if (DebugFlag == TRUE) {
        if (LastData == COMPRESSIONKEY) {
        gotoxy(55, 22);
        cprintf("%8ld", CKCount++);
        }
        else if (LastData == FILLDATA) {
                gotoxy(65, 22);
                cprintf("%#9ld", FDCount++);
                gotoxy(76, 22);
                cprintf("%#3d", CompressCount);
                }
        }
}
else {                          // no
        for (i=0; i<CompressCount; i++) {
        CompressBuffer[(unsigned int)CompressPTR++] = LastData;
        }
        }
        // Is the CompressBuffer close enough to being full?
        if (CompressPTR >= (LocalMaxSectors*512)-6) {
```

```
                error = FlushCompressBuffer();// then flush it to image file
                CompressPTR = 0;
        }
        return error;
}
/*************************************************************************
        Program:  FlushCompressBuffer
        Description: This function flushes the CompressBuffer to the image
        file.
        Returns ZERO on success, nonZero on failure.
        Author: Kevin J. Turpin        Date: 6 May  1996
*************************************************************************/
int
FlushCompressBuffer( void)
{
        // Write the contents of the CompressBuffer to disk
return ( ( write(ImageFileHandle, CompressBuffer, (unsigned
        int)CompressPTR)
        == (unsigned int)CompressPTR)
        ? 0             /* SUCCESS */
        : 1);     /* ERROR */
}
/*************************************************************************
        Program:  DownloadImage
Description: This function performs the downloading of the image (drive
                data).  If a filename was specified by the operator, it is
                opened and the drive data is read from the file.
                If the broadcast feature is enabled, the data is broadcasted
                to the image slaves on the network.
```

```
        Author: Kevin J. Turpin        Date: 15 May 1996
*************************************************************************/
void DownloadImage( void)
{
    // Initialize needed data
    HeadBufferPTR = 0;                  // empty HeadBuffer to start
    // Open the image file if needed and read the image header.
    if (ImageFileNameValid == FALSE) {
        /* We CANNOT download without a valid image file name. */
        sprintf(ErrorMessage, "\nNeed filename for download");
        DisplayLargeMsg(1, "  Error");
        LogError(NEEDFILENAMEERR, ErrorMessage);
        CleanExit();
    }
    if ((ImageFileHandle = open(ImageFileName,
                                O_RDONLY|O_BINARY,
                                S_IREAD)) == -1) {
        sprintf(ErrorMessage, "\nError opening image file %s",
            ImageFileName);
        DisplayLargeMsg(1, "  Error");
        LogError(FILEOPENERR, ErrorMessage);
        CleanExit();
    }
    // Read the image file header and check for proper drive geometry
    if ( read(ImageFileHandle, &ImageFileHeader,
        sizeof(ImageFileHeader))
        != sizeof( ImageFileHeader)) {
    sprintf(ErrorMessage, "\nError reading image file %s",
        ImageFileName);
```

```
                DisplayLargeMsg(1, "  Error");
                LogError(READERR, ErrorMessage);
                CleanExit();
        }
        CheckGeometry();
        // If broadcast enabled, lets pause and let operator start process
        if (BroadcastEnableFlag == TRUE){
                DisplayLargeMsg(1, "  PAUSING");
                printf("\n\nPress any key to start the download and
                        broadcast process...");
                BroadcastDriveGeometry();//soslaves candetermine ifimagefits
        }
        // Now process the desired part of the drive.
        DisplayProcessingScreen();
        // Get ready to start and lets start
#if defined( IGNORE_ORIGINAL_GEOMETRY)
        CurrentCylinder = 1 + ImageFileHeader.partitionStartCylinder;
#else
        CurrentCylinder = ImageFileHeader.partitionStartCylinder;
#endif
        CurrentHead = 0;
        CompressPTR = LocalMaxSectors*512+10;// make it bigger to make
        BytesInCompressBuffer = LocalMaxSectors*512;
                // the program fill the buffer
                // later.
        do {
                ImageDataByte=GetByteFromCompressBuffer();
                if (ImageDataByte == COMPRESSIONKEY) {
                        CompressCount = GetByteFromCompressBuffer();
```

```
                                        // unpack the two remaining
                                ImageDataByte = GetByteFromCompressBuffer();
                                        // bytes of compressed data
                                WriteDataToHeadBuffer(CompressCount, ImageDataByte);
                        }
                        else {                                  // non compressed data
                                WriteDataToHeadBuffer(1, ImageDataByte);
                                                                // only one byte
                        }
//                      if (kbhit()) {
//                              if (toupper(getch()) == 'Q')
//                                      CleanExit();
//                      }
        } while((BytesInCompressBuffer != EOF) && (BytesInCompressBuffer > 0));
#if defined( IGNORE_ORIGINAL_GEOMETRY)
                inform( stderr, "\n\nStartCylinder = %u\n", ImageFileHeader.partitionStartCylinder);
        inform( stderr, "At end, CurrentCylinder is %u, CurrentHead is %u\n",
                CurrentCylinder, CurrentHead);
                inform( stderr, "PAKTC: ");
                (void)getch();
                inform( stderr, "\n");
#endif
#if 0
                // If in broadcast mode, send exit command
                if (BroadcastEnableFlag == TRUE) {
                        OurHeader.command = EXIT;
                        for (i=0; i<3; i++)
                                // multiple times so were sure they get it
                                SendOurIPXPacket();
```

```
        }
#endif
        // Close the image file if used
        if (ImageFileNameValid == TRUE) {
                if ( close(ImageFileHandle)) {
sprintf(ErrorMessage, "\nError closing image file %s", ImageFileName);
        DisplayLargeMsg(1, "  Error");
        LogError(CLOSEERR, ErrorMessage);
        CleanExit();
        }
}
        /*****************************************/
        if ( BroadcastEnableFlag) {
                finishBroadcastProcessing();/* Resendany missed tracks, then
                                             * say "goodbye" to the clients.
                                             */
        }
}
/*************************************************************************
        Program:  GetByteFromCompressBuffer
Description:  This function retrieves a byte from the CompressBuffer.
                When the buffer is empty, it is refilled by reading
                data from the image file.
        Author: Kevin J. Turpin        Date: 16 May  1996
*************************************************************************/
unsigned char GetByteFromCompressBuffer( void)
{
        // if we are empty, fill it up.
        if (CompressPTR >= BytesInCompressBuffer) {
```

```
                FillCompressBufferFromFile();
                CompressPTR = 0;
        }
        return (CompressBuffer[(unsigned int)CompressPTR++]);
}
/*************************************************************************
        Program:  FillCompressBufferFromFile
        Description:  This function fills the CompressBuffer by reading
                      a head size (MaxSectors * 512) of data from the image
                      file.  If an error occurs during the read, an error
                      message is displayed.
        Author: Kevin J. Turpin        Date: 16 May  1996
*************************************************************************/
void FillCompressBufferFromFile( void)
{
        BytesInCompressBuffer = read(ImageFileHandle,
                                     CompressBuffer,
                                     LocalMaxSectors*512);
        if (BytesInCompressBuffer == -1) {
                sprintf(ErrorMessage, "\nError reading image file %s",
                        ImageFileName);
                LogError(READERR, ErrorMessage);
                CleanExit();
        }
}
/*************************************************************************
        Program:  WriteDataToHeadBuffer
Description:  This function takes the byte of data passed to it and
              writes it to the HeadBuffer compressCount times.  This
```

```
                is part of the process of unpacking or uncompressing
                the image file (Run Length compression algorithm).
                When the HeadBuffer is full, we will flush it to the
                hard drive.
        Author: Kevin J. Turpin        Date: 16 May  1996
*****************************************************************/
void WriteDataToHeadBuffer(int compressCount, unsigned char data)
{
        int i;
        // Temp test (kevin)
        if (data != FILLDATA && ValidDataInHead != TRUE)
                ValidDataInHead = TRUE;
        // Put the data in the HeadBuffer
        for (i=0; i<compressCount; i++) {
                HeadBuffer[(unsigned int)HeadBufferPTR++] = data;
                if (HeadBufferPTR >= LocalMaxSectors*512) {
                        FlushHeadBuffer();
                        HeadBufferPTR = 0;
                }
        }
}
/*****************************************************************
        Program:  FlushHeadBuffer
        Description:    This function flushes the HeadBuffer to disk.  The
                entire Head is writen with one command.  All needed
                pointers, counters, etc are updated along with screen
                info.
                Also, if the broadcast feature is enabled, we will
                broadcast the head to the slaves.
```

```
        Author: Kevin J. Turpin        Date: 16 May 1996
    Mod 1:  Added check for valid data before flushing head buffer.
                    Kevin Turpin 12-17-1996
    ************************************************************************/
5   void FlushHeadBuffer( void)
    {
        // Let the operator know where we are
        DisplayProcData();
    // Now lets write the HeadBuffer to disk if it contains valid data.
10      // Note that the ForceWriteData flag can override and force us to
        // always write the data.
        if (ValidDataInHead == TRUE || ForceWriteData == TRUE)
            WriteHeadBufferToDrive();
        /* Update "valid up to" numbers. */
15      ValidUpThruCylinder = CurrentCylinder;
        ValidUpThruHead = CurrentHead;
        // If in broadcast mode, broadcast the head
        if (BroadcastEnableFlag == TRUE) {
            BroadcastHead();
20      }
        // Better update pointers/counters/etc
        if (++CurrentHead > LocalMaxHeads) {
            CurrentHead = 0;
            CurrentCylinder++;
25      }
        ValidDataInHead = FALSE;
        // Get ready for next head of data
    }
    /************************************************************************
```

```
        Program:  WriteHeadBufferToDrive
        Description:  This function writes the HeadBuffer to disk at the
                      CurrentCylinder and CurrentHead.  The data is then
                      read back from the drive and compared with the original
                      data (read after write verify function).
                      If errors are encountered, the process is retried 5 times
                      in an attempt to recover from the error.  It we do not
                      recover, an error message is displayed.
        Author: Kevin J. Turpin      Date: 16 May  1996
**************************************************************************/
void WriteHeadBufferToDrive( void)
{
        int ccode,
            retryCount;
        if ( DebugFlag == TRUE) {
            static unsigned long writeCount = 0L;
            int x=wherex(), y=wherey();
            writeCount++;
            gotoxy( 1, 24);
            cprintf("WriteHeadBufferToDrive count: %lu ", writeCount);
            gotoxy( x, y);
        }
        retryCount = 0;
        do {
            // Try to write the head
            ccode = biosdisk(WRITE,
                    CurrentHDDrive,
                    CurrentHead,
                    CurrentCylinder,
```

```
                    1,              //since we read an entire head, sector=1
                    LocalMaxSectors,
                    HeadBuffer);
                if (ccode) {
                    DisplayBiosError( ccode, 20, 25);
                    retryCount++;
                }
            } while ( ccode && retryCount < 5);
            if (ccode) {                    //are we clean yet? No erros.
                if (ccode != 0x11) {        //we can recover from CRC errors
                    sprintf(ErrorMessage,
                    "\n\nBios error %d writing to cylinder %d, head %d",
                    ccode,
                    CurrentCylinder,
                    CurrentHead);
                LogError(BIOSERR, ErrorMessage);
                CleanExit();
            }
        }
        // Now that we have written it to the drive, read it back and compare
        // But we will only do this if the operator asks us to.
        if (ReadAfterWriteVerifyFlag == TRUE) {
            ReadDriveHead(CompareHeadBuffer);
        if (memcmp(CompareHeadBuffer, HeadBuffer, LocalMaxSectors*512) != 0) {
            sprintf(ErrorMessage,
            "\nRead after write verify error on Cylinder %ld, Head %d",
                                        CurrentCylinder,
                                        CurrentHead);
```

```
                    LogError(VERIFYERR, ErrorMessage);
                CleanExit();
                }
            }
    }
/*************************************************************************
        Program:  CheckGeometry
Description:  This function is called by direction of a received IPX
              packet command.  It takes the geometry received in the
              IPX packet and compares it with the local drive geometry.
              If they are not the same, an error message is displayed
              and the program is terminated.
              If they are the same, the function returns normally.
        Author: Kevin J. Turpin      Date: 6 May  1996
*************************************************************************/
void CheckGeometry( void)
{
#if defined( IGNORE_ORIGINAL_GEOMETRY)
        ImageFileHeader.maxCylinders = LocalMaxCylinders;
        ImageFileHeader.maxHeads = LocalMaxHeads;
        ImageFileHeader.maxSectors = LocalMaxSectors;
        return;
#endif
        if ((ImageFileHeader.maxCylinders != LocalMaxCylinders) ||
                (ImageFileHeader.maxHeads != LocalMaxHeads) ||
                (ImageFileHeader.maxSectors != LocalMaxSectors)) {
        // If the heads and sectors are the same and the image cylinders
        // is less than the local drive cylinders, we can still image it.
```

```
        if ((ImageFileHeader.maxHeads == LocalMaxHeads) &&
                (ImageFileHeader.maxSectors == LocalMaxSectors) &&
                (ImageFileHeader.maxCylinders <= LocalMaxCylinders))
                return;
        sprintf(ErrorMessage, "Wrong Geometry");
        LogError(BADGEOMETRY, ErrorMessage);
        Beep();
        Beep();
        Beep();
        printf("\n\nImage geometry does not match local geometry!");
        printf("\n\nCannot receive the image.");
        printf("\n\nLocal geometry = Cylinder - %d", LocalMaxCylinders);
        printf("\n          Head - %d", LocalMaxHeads);
        printf("\n          Sectors - %d", LocalMaxSectors);
        printf("\n\nImage geometry = Cylinder - %d",
                ImageFileHeader.maxCylinders);
        printf("\n          Head - %d", ImageFileHeader.maxHeads);
        printf("\n          Sectors - %d\n\n",
                ImageFileHeader.maxSectors);
        CleanExit();
        }
}
/*************************************************************************
        Program: Beep
        Description:  This function makes the speaker beep.
        Author: Kevin J. Turpin        Date: 6 May 1996
*************************************************************************/
void Beep( void)
{
```

```
        sound(600);
        tenthMilliDelay(3000);
        nosound();
    }
/*************************************************************************
        Program:  DisplayUsage
        Description:   This function displays the usage for this program.
        Author: Kevin J. Turpin        Date: 13 May  1996
*************************************************************************/
void DisplayUsage( void)
{
        clrscr();
        printf("\nImageBlaster   version %u.%u.%u",
                ProgramMajorVersion,
                ProgramMinorVersion,
                ProgramRevision);
        printf("\nCopyright 1996-1997,  KeyLabs, Inc.  All rights
                reserved.");
        printf("\n"
                "\n"
                "KeyLabs, Inc.\n"
                "633 South 550 East\n"
                "Provo, Utah 84606\n"
                "Phone: 801-377-5484\n");
        /* Print distributor info. */
        decrypt_printstring( distributedBy);
        printf("\n");
        printf(         "\nUSAGE:");
        printf(         "\n  IMGBLSTR [switches]");
```

```
        printf(         "\n");
        printf(         "\nswitches = -U   Upload image");
        printf(         "\n        -D   Download image");
        printf(         "\n        -Px  Partition to process
5               (x=[1..4], 5=entire disk)");
        printf(         "\n        -I[y] y = Image file name
                (optional for broadcast uploads)");
        printf(         "\n        -R   Read after write verify
                on.");
10      if ( BROADCAST_ENABLED()) {
        printf( "\n        -B[n] Broadcast image data to all image
                slaves");
        printf( "\n             (Optional 'n' is number of clients expected.");
        printf( "\n             '-B0' DISables broadcast
15              mode.)");
        }
        printf(         "\n        -Gz  Prepare Drive (z=['A'..'Z'])
                ('G'et Ready for Upload)");
        printf(         "\n        -NT  No physical I/O for Prepare
20              (allows -G in NT DOS box)");
        printf(         "\n        -L   Display License
                information.");
        printf(         "\n");
    /* PROGRAMMERS TAKE NOTE! The following UNDOCUMENTED OPTIONS are
25          recognized
    * by the program but are NOT shown in the 'usage' message:
    *
    *               -z!             Turns on DebugFlag variable.
    *               -db             Turns on "debug broadcast" mode (DebugBroadcast
```

```
        variable)
*               -m              Turns on "monitor broadcast delays" mode
*                                       (MonitorBroadcastDelays variable)
*               -f              Forces ALL data to be written to disk ("empty"
        heads NOT skipped)
*                                       (ForceWriteData variable)
*/
#if 0   /* This is out of place here (displaces rest of messages). */
        // Write Eval only copy
        WriteEvalMessage();
#endif
}
/*************************************************************************
        Program:  DisplayBiosError
Description:    This function displays the error message associated with
                        the BIOS error (as specified in the BC library book).
        Author: Kevin J. Turpin         Date: 6 May  1996
*************************************************************************/
void DisplayBiosError(int ccode, int x, int y)
{
        gotoxy(x, y);
        switch(ccode) {
        case 0x00:
                cprintf("Error %x - Operation successful.", ccode);
                break;
        case 0x01:
                cprintf("Error %x - Bad command.", ccode);
                break;
        case 0x02:
```

```
            cprintf("Error %x - Address mark not found.", ccode);
            break;
        case 0x03:
            cprintf("Error %x - Attempt to write to write-protected
                disk.", ccode);
            break;
        case 0x04:
            cprintf("Error %x - Sector not found.", ccode);
            break;
        case 0x05:
            cprintf("Error %x - Reset failed.", ccode);
            break;
        case 0x06:
            cprintf("Error %x - Disk changed since last operation.",
                ccode);
            break;
        case 0x07:
            cprintf("Error %x - Drive parameter activity failed.",
                ccode);
            break;
        case 0x08:
            cprintf("Error %x - DMA overun.", ccode);
            break;
        case 0x09:
            cprintf("Error %x - Attempt to DMA across 64K boundary.",
                ccode);
            break;
        case 0x0a:
            cprintf("Error %x - Bad sector detected.", ccode);
```

```
            break;
        case 0x0b:
            cprintf("Error %x - Bad track detected", ccode);
            break;
        case 0x0c:
            cprintf("Error %x - Unsupported track.", ccode);
            break;
        case 0x10:
            cprintf("Error %x - Bad CRC/ECC on disk read.", ccode);
            break;
        case 0x11:
            cprintf("Error %x - CRC/ECC corrected data error.", ccode);
            break;
        case 0x20:
            cprintf("Error %x - Controller has failed.", ccode);
            break;
        case 0x40:
            cprintf("Error %x - Seek operation failed.", ccode);
            break;
        case 0x80:
            cprintf("Error %x - Attachment failed to respond.", ccode);
            break;
        case 0xAA:
            cprintf("Error %x - Drive not ready.", ccode);
            break;
        case 0xBB:
            cprintf("Error %x - Undefined error occurred.", ccode);
            break;
        case 0xCC:
```

```
                cprintf("Error %x - Write fault occurred.", ccode);
                break;
        case 0xE0:
                cprintf("Error %x - Status error.", ccode);
                break;
        case 0xFF:
                cprintf("Error %x - Sense operation failed.", ccode
                );
                break;
        default:
                break;
        }
}
/*************************************************************************
        Program: LogError
Description:    This function displays error messages passed to it from
                the calling function. It displays in large text either
                "SUCCESS" or "ERROR" depending on the error code.
        Author: Kevin J. Turpin         Date: 7 May 1996
*************************************************************************/
void LogError(int errCode, char * errMessage)
{
        if (errCode == 0) {
                DisplayLargeMsg(1, "Success");
                printf("\n%s -> Code = %d\n", errMessage, errCode);
        }
        else {
                DisplayLargeMsg(1, "  Error");
                printf("\n%s -> Code = %d\n", errMessage, errCode);
```

```
//                exit(0);
        }
}
/*************************************************************************
        Program:  DisplayStaticScreenData
Description:    This function displays the static portion of the screen
                        text.
        Author: Kevin J. Turpin         Date: 7 May  1996
*************************************************************************/
void DisplayStaticScreenData( void)
{
        int i;
        int originalColor = Text_Color( LIGHTGRAY);
        // ----- Display the static portions of the screen -----
        for (i=1; i<MAX_STATIC_SCREEN_DATA; i++) {
                if ( ! StaticScreenData[ i].broadcastRelated
                                || BroadcastEnableFlag == TRUE) {
                        gotoxy(StaticScreenData[i].x,
                                StaticScreenData[i].y);
                        cprintf("%s", StaticScreenData[i].Text);
                }
        }
        if (DebugFlag == TRUE) {
                gotoxy(63, 18);
                cprintf("Debug Info");
                gotoxy(63, 19);
                cprintf("----------");
                gotoxy(59, 20);
                cprintf("CK      FD     CC");
```

```
                gotoxy(59, 21);
                cprintf("--     --     --");
        }
        Text_Color( originalColor);
}
/*************************************************************************
        Program:  CleanExit
Description:  This function puts the computer back into the proper state
                before exiting.  This includes:
                        - Changing text color back to normal
                        - Freeing any allocated memory
                        - exiting
        Author: Kevin J. Turpin         Date: 7 May  1996
*************************************************************************/
void CleanExit( void)
{
        // Change text back to normal
        Text_Color(LIGHTGRAY);
        // Free any allocated memory
        DeallocateBuffers();
#if 0   /* Commented-out 1/8/1997 by Chris Clark: Causes
         * slaves to get hosed.
         */
// Ifwe arein broadcastmode, send an exit packet to tell slaves bye bye
        if (BroadcastEnableFlag == TRUE) {
                OurHeader.command=EXIT;
                SendOurIPXPacket();
        }
#endif
```

```
        /* Cancel listen packets. */
        if ( BroadcastEnableFlag == TRUE) {
                ShutdownBroadcast();
        }
        /* Close our local socket, if open. */
        if ( LocalSocket != 0) {
                IPXCloseSocket( LocalSocket);
        }
        // Now lets exit
        exit(-1);
}
/*************************************************************************
        Program:  DisplayLocalGeometry
        Description:   This function displays the local drive geometry.
        Author: Kevin J. Turpin        Date: 7 May  1996
*************************************************************************/
void DisplayLocalGeometry( void)
{
        int originalColor = Text_Color( WHITE);
        gotoxy(DynamicScreenData[LOCALCYL].x,
                DynamicScreenData[LOCALCYL].y);
        cprintf("%#4d", LocalMaxCylinders);
        gotoxy(DynamicScreenData[LOCALHEAD].x,
                DynamicScreenData[LOCALHEAD].y);
        cprintf("%#4d", LocalMaxHeads);
        gotoxy(DynamicScreenData[LOCALSECT].x,
                DynamicScreenData[LOCALSECT].y);
        cprintf("%#4d", LocalMaxSectors);
        Text_Color( originalColor);
```

```
}
/*************************************************************************
        Program:  DisplayImageGeometry
Description: This function displays the Image geometry. It also displays
                the processing stop or end geometry
        Author: Kevin J. Turpin        Date: 7 May  1996
*************************************************************************/
void DisplayImageGeometry( void)
{
        int originalColor = Text_Color( WHITE);
        gotoxy(DynamicScreenData[MASTERCYL].x,
                DynamicScreenData[MASTERCYL].y);
        cprintf("%#4d", LocalMaxCylinders);
        gotoxy(DynamicScreenData[MASTERHEAD].x,
                DynamicScreenData[MASTERHEAD].y);
        cprintf("%#4d", LocalMaxHeads);
        gotoxy(DynamicScreenData[MASTERSECT].x,
                DynamicScreenData[MASTERSECT].y);
        cprintf("%#4d", LocalMaxSectors);
        // Now display the ending geometry
        Text_Color(LIGHTGRAY);                  // so it doesn't stand out
        gotoxy(DynamicScreenData[STOPCYL].x,
                DynamicScreenData[STOPCYL].y);
        if (DownloadFlag == TRUE)       // get end cylinder from image file
                cprintf("%#4d", ImageFileHeader.partitionEndCylinder);
        else
                cprintf("%#4d", Partition[PartitionNumber].endCylinder);
        gotoxy(DynamicScreenData[STOPHEAD].x,
                DynamicScreenData[STOPHEAD].y);
```

```
        cprintf("%#3d", LocalMaxHeads);
        gotoxy(DynamicScreenData[STOPSECT].x,
                DynamicScreenData[STOPSECT].y);
        cprintf("%#3d", LocalMaxSectors);
        Text_Color( originalColor);
}
/*************************************************************************
        Program:  DisplayProcData
Description: This function displays the processing data.  This includes
                the current Cylinder, Head, and Sector.
        Author: Kevin J. Turpin        Date: 7 May  1996
*************************************************************************/
void DisplayProcData( void)
{
        int originalColor = Text_Color( WHITE);
        gotoxy(DynamicScreenData[PROCCYL].x,
                DynamicScreenData[PROCCYL].y);
        cprintf("%#4d     %#3d     %#3d", CurrentCylinder,
                                                CurrentHead,
                                                CurrentSector);
//      gotoxy(DynamicScreenData[PROCHEAD].x,
//                              DynamicScreenData[PROCHEAD].y);
//      cprintf("%#3d", CurrentHead);
//
//      gotoxy(DynamicScreenData[PROCSECT].x,
//                              DynamicScreenData[PROCSECT].y);
//      cprintf("%#3d", CurrentSector);
        Text_Color( originalColor);
}
```

```
/*************************************************************************

        Program:  DisplayLocalMAC

Description:   This function displays the MAC address of the local
                computer.
               It also displays the image file name.

        Author: Kevin J. Turpin        Date: 7 May  1996

*************************************************************************/
void DisplayLocalMAC( void)
{
        int originalColor = Text_Color( WHITE);
        gotoxy(DynamicScreenData[LOCALMAC].x,
                DynamicScreenData[LOCALMAC].y);
        sprintf(TmpBuffer, "%2.2x%2.2x%2.2x%2.2x%2.2x%2.2x\0",
                                        NetAddress[4],
                                        NetAddress[5],
                                        NetAddress[6],
                                        NetAddress[7],
                                        NetAddress[8],
                                        NetAddress[9]);
        TmpBuffer[12] = NULL;
        cprintf("%s", TmpBuffer);
        // Now display the image file name
        gotoxy(DynamicScreenData[IMAGEFILE].x,
                DynamicScreenData[IMAGEFILE].y);
        if (ImageFileNameValid == TRUE)
                cprintf("%s", ImageFileName);
        else
                cprintf("None");
        Text_Color( originalColor);
```

```
}
/*************************************************************************
        Program:  DisplaySocketNumbers
Description: This function displays the socket numbers for the broadcast
                function.
        Author: Kevin J. Turpin        Date: 7 May 1996
*************************************************************************/
void DisplaySocketNumbers( void)
{
        if ( BroadcastEnableFlag == TRUE) {
                int originalColor = Text_Color( WHITE);
                gotoxy(DynamicScreenData[SENDSOCKET].x,
                        DynamicScreenData[SENDSOCKET].y);
                cprintf("%x", DOWNLOAD_SOCKET_NUMBER);
                gotoxy(DynamicScreenData[RECVSOCKET].x,
                        DynamicScreenData[RECVSOCKET].y);
                cprintf("%x", SWAP_WORD( LocalSocket));
                Text_Color( originalColor);
        }
}
/*************************************************************************
        Program:  Prepare Drive WORKER
Description:  This function actually does the work of preparing a drive.
                (Whereas the PrepareDrive() function asks the user which
                drive and then calls this function with an appropriate
                parameter).
                This function prepares the hard drive for upload by
                writing data to the unused portions of a partition
                (drive or logical drive).  This is done by opening
```

```
                a file and writing data to it until the operating
                system says there is no more room.  We then close the
                file and delete it.
                This mechanism allows the operating system to handle
                all the details about where the unused portion is.
        Author: Kevin J. Turpin        Date: 16, Oct 1996
*************************************************************************/
void PrepareDriveWorker( char driveChar)
{
        FreespaceInfo_t  DriveSpaceInfo;        // chris' structure def
        struct dfree DfreeInfo;
        long            availableSpace;
        long            KCount = 1;
        long            number4KBlocks;
        int             i;
        int                     ccode;
        int             drive;
        int             fileHandle;
        char filename[80];
/**Fill the buffer that will be written to the drive in large chunks **/
        for (i=0; i<4096; i++) {
                FillData[i] = FILLDATA;
        }
/***** Build the path for the file that will prepare the drive ****/
        drive = driveChar - 'A';
        sprintf(filename, "%c:\\prepare.drv", driveChar);
        printf("\nFile name = %s",filename);
/****    We must figure out how much space is on the drive to prep *****/
        /**** This will be used for making the proper bar graph. ****/
```

```
if ( NTFlag == TRUE) {
/* We must AVOID physical disk I/O since we may be running in
 * an NT DOS BOX which disallows any physical disk I/O.
 */
        getdfree(drive+1, &DfreeInfo);
        /* Calculate freespace based on logical information. */
        availableSpace = (  (long) DfreeInfo.df_bsec
                * (long) DfreeInfo.df_sclus
                * (long) DfreeInfo.df_avail);
#if PRINT_FREESPACE_AND_EXIT
    printf("\n\n"
            "bytes/sect   = %lu\n"
            "sect/cluster = %lu\n"
            "avail clust  = %lu\n"
            "availableSpace %lu (sig %ld)\n",
            (long)DfreeInfo.df_bsec,
            (long)DfreeInfo.df_sclus,
            (long)DfreeInfo.df_avail,
            availableSpace,
            availableSpace);
           exit( 0);
#endif  /* PRINT_FREESPACE_AND_EXIT */
    }
    else {
    ccode = get_freespace(drive, &DriveSpaceInfo);
        if (ccode != 0) {
    printf("\n\nError getting drive space info. (error %d)\n", ccode);
    printf("\n\nDRIVE MAY NOT EXIST!\n\n");
    exit(1);
```

```
    }
        /* Calculate freespace based on physical information. */
        availableSpace = (  (long) DriveSpaceInfo.bytesPerSector
        * (long) DriveSpaceInfo.sectorsPerCluster
        * (long) DriveSpaceInfo.freeClusters);
        }
        number4KBlocks = availableSpace / 4096;
        /****    Now prep the drive by opening a file on the drive and *****/
        /****    filling it until we run out of room.                  *****/
        printf("\n\n");
        (void)unlink(filename);                                          // in case
    it already exists
        if ((fileHandle = open(filename, O_RDWR | O_CREAT | O_BINARY,
                                                    S_IREAD | S_IWRITE )) == -1 ) {
            printf("\n\nError opening %s", filename);
            exit(1);
        }
    /** We are now ready to prepare the drive, lets draw a bar graph ****/
    clrscr();
    _setcursortype(_NOCURSOR);                          // turn cursor off
    DisplayLargeMsg(1, " Preparing");
    gotoxy(31,10);
    cprintf("Preparing Drive %c", driveChar);
    gotoxy(28,12);
    cprintf("Available Space - %ldK", availableSpace/1024);
        // Write Eval only copy
    WriteEvalMessage();
    DrawBarGraph(0);                                        // 0 = new bar graph
    while ((write(fileHandle, FillData, 4096)) == 4096) {
```

```
            KCount = KCount++;
            DrawBarGraph((int)((( KCount - 1)*100) / number4KBlocks));
            }
            /**** We are now done preparing the drive ****/
            /****    Now close the file and delete it!    *****/
    if ( close(fileHandle)) {
            _setcursortype(_NORMALCURSOR);              // turn cursor back on
            printf("\n\nError closing %s", filename);
            exit(1):
    }
    if ( unlink(filename)) {
            _setcursortype(_NORMALCURSOR);              // turn cursor back on
                printf("\n\nError removing %s", filename);
                exit(1);
            }
            _setcursortype(_NORMALCURSOR);      // turn cursor back on
    }
    /*************************************************************************
            Program:  Prepare Drive
    Description:Prompt user for a drive letter, then call
            PrepareDriveWorker()to actually do the work.
            Author: Kevin J. Turpin        Date: 16, Oct 1996
    *************************************************************************/
    void PrepareDrive( void)
    {
            char driveChar;
            clrscr();
            printf("\n\nPlease enter drive letter to prep (A-Z) : ");
                // Write Eval only copy
```

```
        WriteEvalMessage();
        driveChar = toupper(getch());
        if ((driveChar < 'A') || (driveChar > 'Z')) {
                printf("\n\nYou entered an invalid drive letter!");
                exit(1);
        }
        PrepareDriveWorker( driveChar);
}
/*************************************************************************
        Function:       DrawBarGraph(int PercentCompleted)
Description:This function displays a bar graph with a % completednumber.
        Author:  Kevin Turpin                                   Date: 16
Oct, 1996
        Modifications:
*************************************************************************/
void DrawBarGraph(int PercentCompleted)
{
        int i;
        int originalColor = Text_Color( LIGHTGRAY);
        // get to the right place on the screen
        gotoxy(16,18);
        // if % completed = 0, then draw a clean bar graph
        if (PercentCompleted == 0) {
                for (i=0; i<50; i++) {
                        cprintf("%c", 0xB0);        // empty box = 0xB0
                }
        }
        // if other than 0, draw one box for each 2%
        else {
```

```
                Text_Color(YELLOW);
                for (i=0; i<(PercentCompleted/2); i++) {
                        cprintf("%c", 219);          // filled box = 219 dec
                }
        }
        // now go to the right place to display the % value
        Text_Color(LIGHTGRAY);
        gotoxy(33,20);
        cprintf("%3d% Completed", PercentCompleted);
        Text_Color( originalColor);
}
/*************************************************************************
        Function:       WriteEvalMessage
Description:    This function displays an Evaluation Message to inform
                the user that this copy of the utility is an Eval only
                copy.  This message is only displayed if the
                evalProgram flag is set.
        Author: Kevin Turpin                                    Date: 30
Oct, 1996
        Modifications:
*************************************************************************/
void WriteEvalMessage( void)
{
                // Find out if we should display this message
        if ( evalProgram) {
                int xcord, ycord;
                int originalColor = Text_Color( WHITE);
                // Write Eval only copy
                xcord = wherex();
```

```
                ycord = wherey();
                gotoxy(45,23);
                cprintf("*** For Evaluation Purposes Only ***");
                gotoxy(48,24);
                cprintf("*** Not For Production Use ***");
                Text_Color( originalColor);
                gotoxy(xcord,ycord);
            }
        }
/*************************************************************************
        Function:       DetermineNumberOfHardDrives
Description:This function checks to see how many hard drives are in
                the computer.  A default of one hard drive is assumed.
                This function checks for a second hard drive by
                trying to read a section (sector) from the second drive.
                If it is successful, a second drive exists.  If it fails,
                the default of one drive is used for the number of hard
                drives in the computer.
        Author: Kevin Turpin            Date: 30 Oct, 1996
        Modifications:
*************************************************************************/
void DetermineNumberOfHardDrives( void)
{
        int i,
                ccode;
        // We will loop four times to check all IDE possibilities
        // Two drives on primary channel and two on secondary
        // We do this in case there are two drives on seperate channels
        for (i=0; i<4; i++) {
```

```
// Try to read a sector from the second drive
ccode = biosdisk(READ,
                    0x80 + i,                    // drive
                    0,                           // head = 0
                    0,                           // cylinder = 0
                    1,                           // sector=1
                    1,                           // read only one sector
                    HeadBuffer);
if (DebugFlag == TRUE) {
        printf("\nJust tried to check drive ID %x", 0x80 + i);
        printf("\nccode = %d", ccode);
        printf("\nPress any key to continue .....");
        getch();
}
// if successful, number of hard disk drives = 2
if (ccode == 0) {
        if (DebugFlag == TRUE) {
                printf("\n\nI think I found a drive at %x", 0x80 + i);
        }
//Before we assume this is a hard drive, lets try to write the data back
// to it to verify that it is not a CD drive on the same channel.
ccode = biosdisk(WRITE,
                    0x80 + i,                    // drive
                    0,                           // head = 0
                    0,                           // cylinder = 0
                    1,                           // sector=1
                    1,                           // write only one sector
                    HeadBuffer);
if (ccode == 0) {                    // No errors, it was a hard drive
```

```
            if (DebugFlag == TRUE) {
                printf("\n\nI did find a drive at %x", 0x80 + i);
            }
                NumberOfHardDrives++;
                HardDriveID[i] = TRUE;
            // save off id number
                if ( DebugFlag == TRUE) {
            printf("\n\nFOUND a SECOND HARD DRIVE with id 0x%02x. Not looking
                for more.\n",
                0x80 + i);
            printf("Hit any key to continue...");
            (void)getch();
            printf("\n");
            break;
    }
    }
    )
    }
    }
/*************************************************************************
        Function:       AllocateBuffers
    Description:    This function allocates the needed buffers for the
                    program.  These buffers are used for reading and
                    writing to the hard drive.
        Author: Kevin Turpin                Date: 30 Oct, 1996
        Modifications:
*************************************************************************/
void AllocateBuffers( void)
{
```

```
// Allocate memory for Head buffer, compare buffer, compress buffer
if ((HeadBuffer = (unsigned char *) calloc((512*LocalMaxSectors),
        sizeof(char))) == NULL) {
sprintf(ErrorMessage, "Not enough memory to allocate Head buffer!\n");
LogError(ALLOCERR, ErrorMessage);
CleanExit();
}
if ((CompareHeadBuffer = (unsigned char *) calloc((512*LocalMaxSectors),
                                sizeof(char))) == NULL) {
        sprintf(ErrorMessage, "Not enough memory to allocate Compare Head
                buffer!\n");
        LogError(ALLOCERR, ErrorMessage);
        CleanExit();
        }
if ((CompressBuffer = (unsigned char *) calloc((512*LocalMaxSectors),
                                sizeof(char))) == NULL) {
        sprintf(ErrorMessage, "Not enough memory to allocate Compress
                buffer!\n");
        LogError(ALLOCERR, ErrorMessage);
        CleanExit();
        }
}
/*************************************************************************
        Function:       DeallocateBuffers
Description:    This function deallocates the buffers used in the
                        program.
        Author: Kevin Turpin                    Date: 30 Oct, 1996
        Modifications:
*************************************************************************/
```

```
void DeallocateBuffers( void)
{
        if (HeadBuffer != NULL)
                free(HeadBuffer);
        if (CompareHeadBuffer != NULL)
                free(CompareHeadBuffer);
        if (CompressBuffer != NULL)
                free(CompressBuffer);
}
```

```
#define KEY_ENABLE_DEBUG  1        /* Causes '@' key to toggle DebugBroadcast mode. */
#undef BEAUCOUP_VERBOSE            /* Causes VERBOSE packet-processing outputs. */
/*************************************************************************
 * (C) Copyright 1996-1997 KeyLabs, Inc.
 * All Rights Reserved.
 * This program is an unpublished copyrighted work which is proprietary
 * to KeyLabs, Inc. and contains confidential information that is not
 * to be reproduced or disclosed to any other person or entity without
 * prior written consent from KeyLabs, Inc. in each and every instance.
 * WARNING:  Unautorized reproduction of this program as well as
 * unauthorized preparation of derivative works based upon the
 * program or distribution of copies by sale, rental, lease or
 * lending are violations of federal copyright laws and state trade
 * secret laws, punishable by civil and criminal penalties.
 *************************************************************************
/*************************************************************************
        Program:  IMGSLAVE.C
        Description:  This program is the slave component of the parallel disk
                image process.  This slave uses an IPX socket to listen
                for data from the image master.  A special header in this
                data determines the function the slave will perform.
        Author: Kevin J. Turpin        Date: 6 May  1996
        Mod 1:  Added support for multiple receive ECBs.  Kevin  5 Dec 1996
*************************************************************************/
#define         NWDOS
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
```

```
#include <stdarg.h>
#include <string.h>
#include <conio.h>
#include <dos.h>
#include <dir.h>
#include <alloc.h>
#include <bios.h>
#include <time.h>
#include <mem.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <io.h>
#include <process.h>
#include <nwipxspx.h>
#include "imgslave.h"
#include "largemsg.h"
#include "esr.h"
#include "eval.h"
#include "license.h"
/*****************************************************************************/
/*****************************************************************************/
// The next #define's are for controlling how the program works for
// debug, normal.  If the #define DEBUG is uncommented, various debug
// messages will be displayed during execution.
//#define DEBUG
// end of special #define's
/*****************************************************************************/
/* Make sure this program has plenty of stack space
 * (the default is something like 4K).
```

```
 */
unsigned _stklen = 16384;
/*************************************************************************/
/* Space for the License information: */
#include "crypchar.h"      /* Get CRYPT...() macros and decrypt_printstring() proto.
                           */
#define DISTRIB_BY_STRING          \
 CRYPT_16( 'D', 'i', 's', 't', 'r', 'i', 'b', 'u', 't', 'e', 'd', ' ', 'b', 'y', ':', ' ')
char distributedBy[] = {
        DISTRIB_BY_STRING,
        CRYPT_40( 'K', 'e', 'y', 'L', 'a', 'b', 's', ',', ' ', 'I',        \
                'n', 'c', '.', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', \
                ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', \
                ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ),
        0       /* NUL-terminator. */
};
/*************************************************************************/
#define TRUE                    1
#define FALSE                   0
#define WRITE                   3
#define READ                    2
#define ENDOFFILE       0xFF
#define HD_DRIVE        0x80
#define IPXOPENSOCKETERR        -1
#define ALLOCERR                        -2
#define BIOSERR                         -3
#define SECTORERR                       -4
#define BADGEOMETRY                     -5
#define VERIFYERR                       -8
```

```
#define  INCOMPLETE                    -9
#define  ADD                           1      // constants for passing options
#define  NEW                           2
#define  OLD                           3
#define  CLEAR                         4
#define  TEST                          5
#define  INITIALIZE                    6
#define  INCOMPLETE_HEAD               7
#define  COMPLETE_HEAD                 8
#define  NUMBER_ECBs       65          /* One per sector, plus one more so we can
                                        * receive an entire head plus the "exit"
                                        * command (in finish phase).
                                        */
/*****************************************************************************/
/* Handy macros. */
#define ASSERT( x)       assert( x)
/*****************************************************************************/
//---------------------- DECLARATIONS ----------------------------
#include "prog_id.h"
static unsigned int              ProgramID                                    =
IMAGESLAVE_ID;       /* From prog_id.h */
static unsigned int              ProgramMajorVersion     = 1;
static unsigned int              ProgramMinorVersion     = 2;
static unsigned int              ProgramRevision         = 4;
static int          evalProgram = 0; /* Set when only license is eval. */
int                                  socketOpen = 0;
WORD                             LocalSocket = SWAP_WORD( DOWNLOAD_SOCKET_NUMBER);
IPXAddress          serverAddress;
ECB                              receiveECB[NUMBER_ECBs];
```

```
IPXHeader       receiveHeader[NUMBER_ECBs];
BufferPacketHeader *pOurHeader;
BYTE                    NetAddress[12],
IPXDataBuffer[NUMBER_ECBs][512+sizeof( *pOurHeader)];
int     i,
        value,
        HeadComplete = TRUE,
        MasterMACFound = FALSE,
        ImageGeometryFound = FALSE,
        DebugBroadcast = FALSE;
enum {
        STATE_FIND_MASTER,
        STATE_READY,
        STATE_RECEIVING,
        STATE_FINISHING,
        STATE_DISCONNECT
} programState;
int     LocalMaxHeads,
        LocalMaxCylinders,
        LocalMaxSectors,
        LocalNumHeads,
        LocalNumCylinders,
        LocalNumSectors,
        LastRecvdSector = 0,
        IHCount = 0;
long    StartCylinder;    /* First and last cylinders in the image. */
long    EndCylinder;
char    ErrorMessage[80];
unsigned char * CompareHeadBuffer;
```

```
typedef struct ICHeads
{
        int     head;
        long    cylinder;
        struct ICHeads *next;
} ICHEADLIST, *ICHEADPTR;
ICHEADLIST *FirstICHead, *LastICHead;
typedef struct {
        long cylinder;
        long head;
        BYTE *sectorMap;
#define SLOT_EMPTY              0x00
#define SLOT_VALID_DATA     0x01
        BYTE *sectorData;
} TrackBuffer;
#define NUM_TRACKBUFS       1
/*********************************************/
typedef struct {
        long cylinder;
        long head;
} TrackSpec;
/************************/
int
CHEqual( long cyl1, long head1,
        long cyl2, long head2)
{
        return (cyl1 == cyl2 && head1 == head2);
}
/************************/
```

```
int
CHlessThan( long cyl1, long head1,
        long cyl2, long head2)
{
    return (   ( cyl1 < cyl2)
            || (   cyl1 == cyl2
            && head1 < head2));
}
/*******************************************/
struct DynamicScreenType
{
    int   x;
    int   y;
} DynamicScreenData[] = {{ 24,  3+8},     // Local MAC addr
                         { 65,  3+8},     //Master MAC addr
                         { 24,  7+8},     //Local Cyl
                         { 24,  8+8},     //Local Head
                         { 24,  9+8},     //Local Sector
                         { 64,  7+8},     //Image Cyl
                         { 64,  8+8},     //Image Head
                         { 64,  9+8},     //Image Sector
                         { 29, 15+7},     //Proc Cyl
                         { 40, 15+7},     //Proc head
                         { 49, 15+7}};    //Proc sector
#define  LOCALMAC        0
#define  MASTERMAC       1
#define  LOCALCYL        2
#define  LOCALHEAD       3
#define  LOCALSECT       4
```

```
#define  MASTERCYL        5
#define  MASTERHEAD       6
#define  MASTERSECT       7
#define  PROCCYL          8
#define  PROCHEAD         9
#define  PROCSECT         10
#define NUM_DYNAMIC_SCREEN_DATA ( sizeof( DynamicScreenData) / sizeof( struct
DynamicScreenType))
struct StaticScreenType
{
        int  x;
        int  y;
        char * Text;
} StaticScreenData[] = {{ 33,  1+8, "I M G S L A V E"},
                        { 5,  3+8, "Local MAC Address"},
                        { 45,  3+8, "Master MAC Address"},
                        { 10,  5+8, "Local Drive Geometry"},
                        { 50,  5+8, "Image Drive Geometry"},
                        { 10,  6+8, "--------------------"},
                        { 50,  6+8, "--------------------"},
                        { 13,  7+8, "Cylinders"},
                        { 53,  7+8, "Cylinders"},
                        { 13,  8+8, "Heads"},
                        { 53,  8+8, "Heads"},
                        { 13,  9+8, "Sectors"},
                        { 53,  9+8, "Sectors"},
                        { 34, 11+7, "Processing"},
                        { 34, 12+7, "----------"},
                        { 27, 13+7, "Cylinder"},
```

```
                { 27, 14+7, "--------"},
                { 39, 13+7, "Head"},
                { 39, 14+7, "----"},
                { 47, 13+7, "Sector"},
                { 47, 14+7, "------"} };
#define NUM_STATIC_SCREEN_DATA ( sizeof( StaticScreenData) / sizeof( struct StaticScreenType))
//------------------- Function Declarations -----------------------------
void Initialize( void);
void GetDriveGeometry( void);
void CheckGeometry( GeometryPacket *pGeom);
void Beep( void);
void DisplayUsage( void);
void DisplayBiosError(int ccode, int x, int y);
void LogError(int errCode, char * errMessage);
void DisplayStaticScreenData( void);
void CleanExit( void);
void DisplayLocalGeometry( void);
void DisplayImageGeometry( GeometryPacket *pGeom);
void DisplayProcData( long cylinder, long head, long sector);
void DisplayMasterMAC( ECB far *ecb);
void DisplayLocalMAC( BYTE *netAddr);
void WriteEvalMessage( void);
/*************************************************************************/
/* Inform for DEBUG-BROADCAST mode only. */
void
DB_INFORM( char *format, ...)
{
        static int line = 1;
        int x, y;
```

```
        va_list   args;
        if ( DebugBroadcast) {
                x = wherex();
                y = wherey();
                textcolor(LIGHTGRAY);
                gotoxy( 1, line);
                clreol();
                va_start( args, format);
                vprintf( format, args);
                fflush( stdout);
                line++;
                if ( line > 7) {
                        line = 1;
                }
                gotoxy( 1, line);
                clreol(); /* Clear NEXT line. */
                gotoxy( x, y);      /* Restore original cursor. */
                textcolor(WHITE);
        }
}
/*************************************************************************/
/* Shim function to Display a "LargeMessage" at the top of the screen
 * without disrupting the current cursor position.
 */
void
TopLargeMsg( int clearScreenFlag, char *msg)
{
        int x = wherex();
        int y = wherey();
```

```
        gotoxy( 1, 1);
        DisplayLargeMsg( clearScreenFlag, msg);
        gotoxy( x, y);
    }
/*************************************************************************/
void
inform( FILE *output_stream, char *format, ...)
{
        va_list   args;
        va_start( args, format);
        vfprintf( output_stream, format, args);
        fflush( output_stream);
}
/*************************************************************************/
/* Wait for a packet to be received on the given socket.
 * If terminateOnKeystroke is nonZero, we return nonZero when THAT key is pressed.
 * If timeoutTicks is nonZero, we return nonZero when that many ticks have
 * elapsed.
 * Otherwise, we wait for a packet to be received, and return ZERO
 * when the packet is successfully received.
 */
int
waitForPacket( int terminateOnKeystroke,
                clock_t timeoutTicks,
                WORD socket,
                int fragCount,
                ECBFragment *frags,
                BYTE *ccode)
{
```

```
ECB     receiveECB;
int frag;
clock_t endTime = (timeoutTicks == 0) ? 0 : clock() + timeoutTicks;
receiveECB.ESRAddress = NULL;
receiveECB.socketNumber = socket;
receiveECB.fragmentCount = fragCount;
for ( frag = 0 ; frag < fragCount ; frag++) {
        receiveECB.fragmentDescriptor[ frag].address = frags[ frag].address;
        receiveECB.fragmentDescriptor[ frag].size = frags[ frag].size;
}
IPXListenForPacket( &receiveECB);
/* Wait for a packet to come in, or, optionally,
 * a keypress, or, optionally,
 * a timer expiration.
 */
while ( 1) {
        IPXRelinquishControl();
        if ( receiveECB.inUseFlag == 0) {
                /* Got a packet.  Return ZERO. */
                *ccode = receiveECB.completionCode;
                return 0;
        }
        else if ( terminateOnKeystroke && kbhit()) {
                /* Key was pressed.  Cancel the listen then
                 * gobble keystroke(s) and return nonZero.
                 */
                int choice = getch();
                if ( choice == 0) {
                        /* two-character keystroke. */
```

```
                        (void)getch();      /* Throw away second byte -- don't match 2-char keys. */
                        }
                        else if ( choice == terminateOnKeystroke) {
                                IPXCancelEvent( &receiveECB);
                        return 1; /* Return nonZero: key was pressed and NO packet received. */
                        }
#if defined( KEY_ENABLE_DEBUG)
                        else if ( choice == '@') {
                                if ( DebugBroadcast != TRUE) {
                                        DebugBroadcast = TRUE;
                                        DB_INFORM( "DebugBroadcast Enabled.\n");
                                }
                                else {
                                        DB_INFORM( "DebugBroadcast Disabled.\n");
                                        DebugBroadcast = FALSE;
                                }
                        }
#endif
                }
                else if ( timeoutTicks && ( clock() > endTime)) {
                        /* Clock timeout.  Cancel the listen then
                         * return nonZero.
                         */
                        IPXCancelEvent( &receiveECB);
                        return 1; /* Return nonZero: timeout and NO packet received. */
                }
        }
}
```

```
/*************************************************************************/
/* Send an RSVP packet to the server.
 * Return ZERO on success, nonZero on failure.
 */
BYTE
sendRSVPPacket( void)
{
        ECB     sendECB;
        IPXHeader header;
        RSVPPacket      rsvp;
        int transportTime;
        /* Fill in the RSVP packet data
         */
        rsvp.command = RSVP;
        /* Fill in the IPX packet header */
        memmove( &header.destination,
                &serverAddress,
                sizeof( serverAddress));
        header.packetType = 4;
        /* Fill in the ECB. */
        sendECB.ESRAddress = NULL;
        sendECB.socketNumber = LocalSocket;
        sendECB.fragmentCount = 2;
        sendECB.fragmentDescriptor[0].address = &header;
        sendECB.fragmentDescriptor[0].size = sizeof( header);
        sendECB.fragmentDescriptor[1].address = &rsvp;
        sendECB.fragmentDescriptor[1].size = sizeof( rsvp);
        if ( IPXGetLocalTarget( (BYTE far *)&serverAddress,
        sendECB.immediateAddress,
```

```
        &transportTime)) {
                return 0xff;        /* Error. */
        }
        /* Send the packet. */
        IPXSendPacket( &sendECB);
        while ( sendECB.inUseFlag) {
                IPXRelinquishControl();
        }
        return sendECB.completionCode;
}
/*************************************************************************/
/* Send a "fare thee well" packet to the server.
 * Return ZERO on success, nonZero on failure.
 */
BYTE
sendFarewellPacket( void)
{
        ECB     sendECB;
        IPXHeader header;
        FarewellPacket    packet;
        int transportTime;
        /* Fill in the packet data
         */
        packet.command = FAREWELL;
        /* Fill in the IPX packet header */
        memmove( &header.destination,
                &serverAddress,
                sizeof( serverAddress));
```

```
        header.packetType = 4;
        /* Fill in the ECB. */
        sendECB.ESRAddress = NULL;
        sendECB.socketNumber = LocalSocket;
        sendECB.fragmentCount = 2;
        sendECB.fragmentDescriptor[0].address = &header;
        sendECB.fragmentDescriptor[0].size = sizeof( header);
        sendECB.fragmentDescriptor[1].address = &packet;
        sendECB.fragmentDescriptor[1].size = sizeof( packet);
        if ( IPXGetLocalTarget( (BYTE far *)&serverAddress,
                                sendECB.immediateAddress,
                                &transportTime)) {
                return 0xff;        /* Error. */
        }
        /* Send the packet. */
        IPXSendPacket( &sendECB);
        while ( sendECB.inUseFlag) {
                IPXRelinquishControl();
        }
        return sendECB.completionCode;
}
/*********************************************************************/
/* Wait for geometry packet, send back "rsvp" packet,
 * then wait for "rsvp ACK" packet.
 *
 * Return ZERO if we get the ACK.  Return nonZero if we receive an
 * unknown packet. (We expect both GEOMETRY and "RSVP_ACK" packets.)
 */
int
```

```
registerWithServer( void)
{
        ECBFragment     rxFrags[ 2] = {
                { &receiveHeader[ 0], sizeof( receiveHeader[ 0]) },
                { IPXDataBuffer[ 0], sizeof( IPXDataBuffer[ 0]) }
        };
        BYTE ccode;
        GeometryPacket *geomPacket;
        while ( 1) {
                /* Wait for a packet. */
                if ( waitForPacket( 1,      /* Terminate on ^A (control-A) keypresses. */
                                    0,      /* Don't timeout based on the clock. */
                                    LocalSocket,
                                    2,
                                    rxFrags,
                                    &ccode) != 0) {
                        /* No packet received: key was pressed instead. */
                        inform( stderr, "\nGeometry-packet wait cancelled by keypress.\n");
                        exit( 1);
                }
                /* Got a packet. Check the completion code. */
                if ( ccode) {
                        inform( stderr, "\nPacket reception error: ccode 0x%x\n", (0xff & ccode));
                        exit( 1);
                }
                /* Packet received OK.  Get the header. */
                geomPacket = (GeometryPacket *)IPXDataBuffer[ 0];
                switch( geomPacket->command) {
                case GEOMETRY:
```

```
                /* Save the server address. */
                memmove( &serverAddress,
                        &(receiveHeader[ 0].source),
                        sizeof( serverAddress));
                CheckGeometry( geomPacket);
                /* Record the starting and ending cylinders. */
                StartCylinder = geomPacket->firstCyl;
                EndCylinder = geomPacket->lastCyl;
                /* Send back an "RSVP" packet. */
                ccode = sendRSVPPacket();
                if ( ccode) {
                inform( stderr, "\nError sending RSVP packet: ccode 0x%x\n", ccode & 0xff);
                        exit( 1);
                }
                break;
        case CONFIRM_DOWNLOAD:
                /* Hey, the server knows about us. We're in! */
                return 0;
        default:
                if ( ImageGeometryFound) {
                        /* Assume our CONFIRM_DOWNLOAD packet got lost, and the
                         * server has started sending data already, so go ahead
                         * and enter the RECEIVING phase.
                         */
                        return 0;
                }
        DB_INFORM( "Unexpected pkt cmd %d while trying to register for a download.\n",
                                geomPacket->command);
                break;  /* IGNORE it. */
```

```
                }
            }
        }
        /*******************************************/
        /* Return 0xffffffff if no zeroes are found. */
        static const BYTE oneBits[ 8] = {
                0x01 << 0,
                0x01 << 1,
                0x01 << 2,
                0x01 << 3,
                0x01 << 4,
                0x01 << 5,
                0x01 << 6,
                0x01 << 7
        };
        unsigned long
        findZero( BYTE *cylHeadMap, unsigned long numBits)
        {
                unsigned int index;
                unsigned int numBytes = 1 + (int)( numBits / 8);
                for ( index = 0 ; index < numBytes ; index++) {
                        if ( cylHeadMap[ index] != 0xff) {
                                unsigned int bit;
                                for ( bit = 0 ; bit < 8 ; bit++) {
                                        unsigned long bitNumber = ( (  (unsigned long)index
                                                        * (unsigned long)8L)
                                                + (unsigned long)bit);
                                        if ( bitNumber >= numBits) {
                                                return 0xffffffffL;/* Zero-bit not found. */
```

```
                    }
                    if ( ! ( cylHeadMap[ index] & oneBits[ bit])) {
                        /* Found it! */
                        return bitNumber;
                    }
                }
            }
        }
        /* Didn't find any zeroes. */
        return 0xffffffffL;
}
/*****************************************/
/* Return state of a bit in the bitmap. */
int
bitValue( BYTE *bitMap, unsigned long bit)
{
        ASSERT( (bit / 8L) < 65536L);
        return ( ( bitMap[ (unsigned int)( bit / 8)]
                & oneBits[ (unsigned int)( bit % 8)])
                ? 1
                : 0);
}
/*****************************************/
/* Set a bit in a bitmap. */
void
setBit( BYTE *bitMap, unsigned long bit)
{
        ASSERT( (bit / 8L) < 65536L);
        bitMap[ (unsigned int)( bit / 8)] |= oneBits[ (unsigned int)(bit % 8)];
```

```
}
/*************************************************************************/
/* Send packet to server re-requesting a missed track.
 *
 * Returns the BYTE completioncode of the IPX send.
 * (Also returns 0xff if no "missing" track can be found.)
 *
 */
BYTE
sendMissingTrackRequest( long cyl,
                         long head)
{
        ECB sendECB;
        IPXHeader header;
        ResendPacket resend;
        int transportTime;
        /* Fill in the Resend packet data
         */
        /* Immediate Address. */
        if ( IPXGetLocalTarget( (BYTE far *)&serverAddress,
                                sendECB.immediateAddress,
                                &transportTime)) {
                return 0xff;        /* ERROR. */
        }
        /* Packet data: missing cylinder and head. */
        resend.command = RESEND_REQUEST;
        resend.cylinder = cyl;
        resend.head = head;
        /* Fill in the IPX packet header */
```

```
            memmove( &header.destination,
                    &serverAddress,
                    sizeof( serverAddress));
            header.packetType = 4;
            /* Fill in the ECB. */
            sendECB.ESRAddress = NULL;
            sendECB.socketNumber = LocalSocket;
            sendECB.fragmentCount = 2;
            sendECB.fragmentDescriptor[0].address = &header;
            sendECB.fragmentDescriptor[0].size = sizeof( header);
            sendECB.fragmentDescriptor[1].address = &resend;
            sendECB.fragmentDescriptor[1].size = sizeof( resend);
            /* Send the packet. */
            IPXSendPacket( &sendECB);
            while ( sendECB.inUseFlag) {
                    IPXRelinquishControl();
            }
            DB_INFORM( "Sent request for missed track (%ld, %ld)\n",
                                                    cyl, head);
            return sendECB.completionCode;
    }
    /*************************************************************************/
    /* Send packet(s) to server re-requesting missed tracks.
     *
     * Returns the BYTE completioncode of the IPX send.
     * (Also returns 0xff if no "missing" track can be found.)
     *
     */
    BYTE
```

```
sendRequestForMissingTracks( BYTE *cylHeadMap)
{
        unsigned long track;
        unsigned long numTracks;
        unsigned long missingCyl;
        unsigned long missingHead;
        BYTE ccode;
        /*****************************************/
        /* Loop through all tracks looking for missing ones.
         * Send a resend request for the first missing one.
         */
        numTracks = ( 1L + EndCylinder - StartCylinder) * LocalNumHeads;
        for ( track = 0L ; track < numTracks ; track++) {
                if ( ! bitValue( cylHeadMap, track)) {
                        missingCyl = StartCylinder + ( track / LocalNumHeads);
                        missingHead = track % LocalNumHeads;
                        ccode = sendMissingTrackRequest( missingCyl, missingHead);
                        if ( ccode) {
                                DB_INFORM( "srfm_tracks: error sending request for (%ld, %ld).\n",
missingCyl, missingHead);
                                return ccode;
                        }
                        DB_INFORM( "srfm_tracks:Sent request for (%ld, %ld).\n", missingCyl,
missingHead);
                        break;   /* Done! */
                }
        }
        return 0;
}
```

```
/*************************************************************************/
/* Perform download from server.  We process STD_DATA, FLUSH, and EXIT
 * packets from the server.
 *
 * We return ZERO when we have downloaded the entire image from the
 * server.
 * We return nonZero if we failed to download the entire image from
 * the server.
 *
 */
/*************************************************************************/
void
processDataPacket( TrackBuffer *trackInfo,
                   BYTE *cylHeadMap,
                   BufferPacketHeader *pOurHeader,
                   BYTE *data)
{
        int i;
#if defined( CHECK_FOR_OUT_OF_RANGE_CHS)
        if ( pOurHeader->cylinder >= LocalNumCylinders
                        || pOurHeader->head >= LocalNumHeads
                        || pOurHeader->sector > LocalNumSectors) {
                        DB_INFORM( "PDP: Illegal CHS (%ld,%ld,%ld) LNCHS=(%u,%u,%u),
SC=%lu, EC=%lu\n",
                        pOurHeader->cylinder, pOurHeader->head, pOurHeader->sector,
                        LocalNumCylinders,
                        LocalNumHeads,
                        LocalNumSectors,
                        StartCylinder,
```

```
                                        EndCylinder);
                                (void)getch();
            }
    #endif
            /*******************************************/
            /* If we already have this track, we don't need to process it. */
            if ( bitValue( cylHeadMap, ( ( ( pOurHeader->cylinder - StartCylinder)
                                        * LocalNumHeads)
                                        + pOurHeader->head))) {
    #if defined( BEAUCOUP_VERBOSE)
                    /* This is VERY verbose--it prints on EVERY packet.
                     */
                    DB_INFORM( "SKIPPING UNNEEDED (%ld,%ld,%ld) LNCHS=(%u,%u,%u),
    SC=%lu\n",
                     pOurHeader->cylinder, pOurHeader->head, pOurHeader->sector,
                     LocalNumCylinders,
                     LocalNumHeads,
                     LocalNumSectors,
                     StartCylinder);
    #endif
                    return;
            }
            /*******************************************/
            /* See if we have a trackbuf to which we can add
             * this packet data.
             */
            for ( i = 0 ; i < NUM_TRACKBUFS ; i++) {
                    if ( CHEqual( pOurHeader->cylinder, pOurHeader->head,
                                    trackInfo[ i].cylinder, trackInfo[ i].head)) {
```

```
                    break;    /* Found the right track. 'i' contains its slot number. */
                }
            }
            if ( i >= NUM_TRACKBUFS) {
                /* NONE of the existing track buffers is assigned to the
                 * cyl/head specified in the packet.  Any existing
                 * partial track buffers are therefore "partial", so
                 * we request a resend of these tracks.
                 *
                 * Then we find a slot where we can start storing the
                 * "current" track data.  If all the track buffers are full,
                 * we clobber the lowest-numbered track and use that slot.
                 */
                for ( i = 0 ; i < NUM_TRACKBUFS ; i++) {
                    if ( trackInfo[ i].cylinder != -1) {
                        /* Send request for lost track. */
                        if ( sendMissingTrackRequest( trackInfo[ i].cylinder,
                                trackInfo[ i].head)) {
                            DB_INFORM( "sendMissingTrackRequest failure.\n");
                        }
                    }
                }
                /* Now search for an empty slot into which we can start this new
                 * track.
                 */
                for ( i = 0 ; i < NUM_TRACKBUFS ; i++) {
                    if (   trackInfo[ i].cylinder == -1) {
                        break;    /* Found it. */
                    }
```

```
        }
        if ( i >= NUM_TRACKBUFS) {
                /* All track buffers full.  Zap the lowest-numbered
                 * buffer for use NOW--we'll re-request the missing track
                 * so that the server will send it to us again later.
                 */
                long lowest_cyl = 0x7fffffffL;
                long lowest_head = 0x7fffffffL;
                int lowest_index = -1;
                for ( i = 0 ; i < NUM_TRACKBUFS ; i++) {
                        if ( CHlessThan( trackInfo[ i].cylinder, trackInfo[ i].head,
                                                lowest_cyl, lowest_head)) {
                                lowest_cyl = trackInfo[ i].cylinder;
                                lowest_head = trackInfo[ i].head;
                                lowest_index = i;
                        }
                }
                ASSERT( lowest_index != -1);
                i = lowest_index; /* Setup 'i' to point to the "lost" track slot. */
        }
        /* 'i' points to a slot where we can start a new track. */
        trackInfo[ i].cylinder = pOurHeader->cylinder;
        trackInfo[ i].head = pOurHeader->head;
        memset( trackInfo[ i].sectorMap, SLOT_EMPTY, LocalNumSectors);
        DB_INFORM( "Started new track in slot %u for C=%lu, H=%lu.\n",
                i, trackInfo[ i].cylinder, trackInfo[ i].head);
        /* Fall-through to store the data in the track buffer. */
}
/* i points to the trackInfo buffer for this packet. */
```

```
            if (   trackInfo[ i].sectorMap[ (unsigned int)(pOurHeader->sector - 1)]
                        != SLOT_VALID_DATA) {
                    memmove( &trackInfo[ i].sectorData[ 512 * (unsigned int)(pOurHeader->sector - 1)],
                            data,
                            512);
                    trackInfo[ i].sectorMap[ (unsigned int)(pOurHeader->sector - 1)] =
SLOT_VALID_DATA;
            }
#if defined( BEAUCOUP_VERBOSE)
            if ( DebugBroadcast) {
                    static char lineBuf[ 250];
                    int q;
                    int x,y;
                    for ( q = 0 ; q < LocalNumSectors ; q++) {
                            if ( trackInfo[ i].sectorMap[ q] == SLOT_EMPTY) {
                                    lineBuf[ q] = '.';
                            }
                            else {
                                    lineBuf[ q] = '#';
                            }
                    }
                    lineBuf[ q] = '\0';
                    x=wherex();
                    y=wherey();
                    gotoxy( 1, 20);
                    cprintf( "C %ld, h %ld:%s".
                    trackInfo[ i].cylinder,
                    trackInfo[ i].head,
                    lineBuf);
```

```
                clreol();
                gotoxy( x, y);
            }
        #endif
    }
    /*****************************************/
    /* Write a track to disk. */
    void
    writeTrack( TrackBuffer *track)
    {
        int ccode,
                retryCount;
        retryCount = 0;
        do {
            // Try to write the head
            ccode = biosdisk(WRITE,
                    HD_DRIVE,
                    (int)track->head,
                    (int)track->cylinder,
                    1,                  //since we read an entire head, sector=1
                    LocalNumSectors,
                    track->sectorData);
            if ( ccode) {
                DisplayBiosError( ccode, 20, 25);
                retryCount++;
            }
        } while ( ccode && retryCount < 5);
        if (ccode) {
            //are we clean yet? No erros.
```

```
                if (ccode != 0x11) {                                    //we can
recover from CRC errors
                    sprintf(ErrorMessage,
                        "\n\nBios error %d writing at cylinder %d, head %d",
                        ccode,
                        track->cylinder,
                        track->head);
                    LogError(BIOSERR, ErrorMessage);
                    CleanExit();
                }
            }
            /*****************************************/
            /* Write was successful.  Now read the track
             * back and compare to the original.
             */
#if 0
            if ( 0) {
                retryCount = 0;
                do {
                    // Try to write the head
                    ccode = biosdisk(READ,
                        HD_DRIVE,
                        (int)track->head,
                        (int)track->cylinder,
                        1,
                    //since we read an entire head, sector=1
LocalNumSectors,
CompareHeadBuffer);
                    if ( ccode) {
```

```
                    DisplayBiosError( ccode, 20, 25);
                    retryCount++;
                }
            } while ( ccode && retryCount < 5);
            if (ccode) {
                //are we clean yet? No erros.
                if (ccode != 0x11) {
        //we can recover from CRC errors
                    sprintf(ErrorMessage,
                    "\n\nBios error %d reading at cylinder %d, head %d",
                    ccode,
                    track->cylinder,
                    track->head);
                    LogError(BIOSERR, ErrorMessage);
                    CleanExit();
                }
            }
            /* Got the data OK.  Now compare. */
            if ( memcmp( track->sectorData, CompareHeadBuffer, LocalNumSectors * 512)) {
                sprintf(ErrorMessage,
                "\n\nData mismatch error at cylinder %d, head %d",
                        track->cylinder,
                        track->head);
                LogError(BIOSERR, ErrorMessage);
                CleanExit();
            }
        }
#endif
}
```

```
/*****************************************************************************/
/* Update the cylinder/head map (set the bit for the good track),
 * then update the "goodUpTo" value.
 */
void
markTrackComplete( BYTE *cylHeadMap,
                   long cylinder,
                   long head)
{
        unsigned long cylHead;
        cylHead = (  ( cylinder - StartCylinder) * LocalNumHeads
                + head);
        setBit( cylHeadMap, cylHead);
}
/*****************************************************************************/
void
flushCompleteTracks( BYTE *cylHeadMap,
                     TrackBuffer *trackInfo)
{
        int i;
        int sector;
        for ( i = 0 ; i < NUM_TRACKBUFS ; i++) {
                if ( trackInfo[ i].cylinder == -1) {
                        continue;       /* Empty slot. */
                }
                for ( sector = 0 ; sector < LocalNumSectors ; sector++) {
                        if ( trackInfo[ i].sectorMap[ sector] != SLOT_VALID_DATA) {
                                break;
                        }
```

```
                }
                if ( sector >= LocalNumSectors) {
                        /* COMPLETE HEAD */
                        /* Flush it to disk. */
                        writeTrack( &trackInfo[ i]);
                        markTrackComplete( cylHeadMap,
                         trackInfo[ i].cylinder,
                         trackInfo[ i].head);
                        /* Mark track buffer good for reuse. */
                        trackInfo[ i].cylinder = -1;
                        trackInfo[ i].head = -1;
                        memset( trackInfo[ i].sectorMap, SLOT_EMPTY, LocalNumSectors);
                }
        }
}
/*************************************************************************/
void
processSkipTrackPacket( BYTE *cylHeadMap,
                        TrackBuffer *trackInfo,
                        SkipTrackPacket *stp)
{
        int i;
        /*****************************************/
        /* Find any track buffers with the skipped cyl/head and
         * zap the buffers.
         * (This should never happen, really, but we're being paranoid.)
         */
        for ( i = 0 ; i < NUM_TRACKBUFS ; i++) {
                if ( CHEqual( stp->cylinder, stp->head,
```

```
                        trackInfo[ i].cylinder, trackInfo[ i].head)) {
                        trackInfo[ i].cylinder = -1;
                        trackInfo[ i].head = -1;
                        memset( trackInfo[ i].sectorMap, SLOT_EMPTY, LocalNumSectors);
                }
        }
        /*****************************************/
        /* Now mark the track complete in the bitmap and adjust our "goodUpTo"
         * values if needed.
         */
        markTrackComplete( cylHeadMap, stp->cylinder, stp->head);
        DB_INFORM( "SKIP (%ld,%ld)", stp->cylinder, stp->head);
}
/*************************************************************************/
/* Return ONE if we should quit, or ZERO if we should continue. */
int
processPacket( BYTE *cylHeadMap,
                TrackBuffer *trackInfo,
                ECB far *ecb)
{
        IPXHeader *pHdr = ecb->fragmentDescriptor[ 0].address;
        BufferPacketHeader *pOurHeader = ecb->fragmentDescriptor[ 1].address;
        BYTE *data = ((BYTE *)pOurHeader) + sizeof( *pOurHeader);
        /*****************************************/
        /* Validate that packet came from server from which
         * we received geometry info.
         */
        if ( memcmp( &serverAddress, &( pHdr->source), sizeof( serverAddress))) {
                DB_INFORM( "Dropping packet from other Master.");
```

```
            return 0;
        }
        /*****************************************/
        /* Display Master's MAC address on the first received packet. */
        if (MasterMACFound == FALSE) {
            DisplayMasterMAC( ecb);
            MasterMACFound = TRUE;
        }
        /*****************************************/
        /* When first packet received, display a message signifying
         * our transition from "ready" to "receiving" state.
         */
        if (  programState == STATE_READY
                    && pOurHeader->command != GEOMETRY) {
                    programState = STATE_RECEIVING;
                    TopLargeMsg( 0, " RECEIVING ");
        }
        /*****************************************/
        switch( pOurHeader->command) {
        case STD_DATA:
            processDataPacket( trackInfo,
             cylHeadMap,
             pOurHeader,
             data);
            break;
        case FLUSH:
            processDataPacket( trackInfo,
             cylHeadMap,
             pOurHeader,
```

```
                data);
                flushCompleteTracks( cylHeadMap,
                trackInfo);
                DisplayProcData( pOurHeader->cylinder, pOurHeader->head, pOurHeader->sector);
                break;
        case SKIP_TRACK: {
                SkipTrackPacket *stp = (SkipTrackPacket *)ecb->fragmentDescriptor[ 1].address;
                processSkipTrackPacket( cylHeadMap,
                trackInfo,
                stp);
                DisplayProcData( stp->cylinder, stp->head, 0);
                break;
        }
        case EXIT:
                /* Server is done sending data.  We're done only if we
                 * have NO missing fragments.
                 */
                if ( programState != STATE_FINISHING) {
                        programState = STATE_FINISHING;
                        TopLargeMsg( 0, " FINISHING ");
                }
#if 0   /* BIG BAD MISSED SECTOR DEBUG */
if ( programState == STATE_FINISHING) {
        int x=wherex(), y=wherey();
        int i;
        gotoxy( 1, 9);
        cprintf("EXIT"); clreol();
        gotoxy( 1, 10);
        cprintf( "TB: cyl %ld, head %ld\r\n". trackInfo->cylinder, trackInfo->head);
```

```
            for ( i = 0 ; i < LocalNumSectors ; i++) {
                    cprintf( "%c", ( trackInfo->sectorMap[ i] == SLOT_EMPTY) ? '.' : '#');
            }
            gotoxy( x, y);
    }
    #endif
                    /* We can exit only if we have no missing tracks. */
                    if ( findZero( cylHeadMap, ( ( 1 + EndCylinder - StartCylinder)
                                            * LocalNumHeads)) == 0xffffffffL) {
                            DB_INFORM( "processPacket(): No missing tracks, we're done.\n");
                            return 1; /* DONE! */
                    }
                    else {
                            /* We have missing fragments. Send "missing track" packet(s). */
                            if ( sendRequestForMissingTracks( cylHeadMap)) {
                                    DB_INFORM( "sendRequestForMissingTracks failure.\n");
                            }
                            DB_INFORM( "processPacket(): sent Request for missing tracks.\n");
                    }
                    break;
            default:
                    /* Ignore other packet types. (We expect to see GEOMETRY packets
                     * until the server starts really sending data.
                     */
                    DB_INFORM( "processPacket(): Dropped Packet type 0x%02x.\n", 0xff & pOurHeader-
    >command);
                    break;
            }
            return 0; /* Need to continue. */
```

```
}
/*******************************************/
/* Return ZERO on success, nonZero on failure.
 *
 */
int
receiveData( void)
{
        TrackBuffer trackInfo[ NUM_TRACKBUFS];
        BYTE *cylHeadMap;
        unsigned long cylHeads = ( 1 + EndCylinder - StartCylinder) * LocalNumHeads;
        unsigned int bytesInCylHeadMap = 1 + (unsigned int)( cylHeads /8);
        int i;
        int done;
        /*******************************************/
        /* Allocate cylinder/head map. */
        cylHeadMap = (BYTE *)malloc( bytesInCylHeadMap);
        if ( cylHeadMap == NULL) {
                inform( stderr, "\nError allocating track buffers.\n");
                return 1; /* ERROR */
        }
        memset( cylHeadMap, 0x00, bytesInCylHeadMap);
        /*******************************************/
        /* Allocate track buffers. */
        for ( i = 0 ; i < NUM_TRACKBUFS ; i++) {
                trackInfo[ i].cylinder = -1;
                trackInfo[ i].head = -1;
                trackInfo[ i].sectorMap = (BYTE *)malloc( LocalNumSectors);
                if ( trackInfo[ i].sectorMap == NULL) {
```

```
                break;
            }
            /* Initialize sector map. */
            memset( trackInfo[ i].sectorMap, SLOT_EMPTY, LocalNumSectors);
            trackInfo[ i].sectorData = (BYTE *)malloc( LocalNumSectors * 512);
            if ( trackInfo[ i].sectorData == NULL) {
                free( trackInfo[ i].sectorMap);
                break;
            }
        }
        /*******************************************/
        /* If error during allocation, free currently-allocated
         * buffers and return error.
         */
        if ( i < NUM_TRACKBUFS) {
            /* ERROR allocating buffers. */
            for ( --i ; i >= 0 ; i--) {
                free( trackInfo[ i].sectorData);
                free( trackInfo[ i].sectorMap);
            }
            inform( stderr, "\nError allocating track buffers.\n");
            return 1; /* ERROR */
        }
        /*******************************************/
        /* Initialize "completed ECBs" list. */
        initEsrQueue();
        /* Initialize and Post a bunch of receive ECBs */
        for (i=0; i<NUMBER_ECBs; i++) {
            receiveECB[i].ESRAddress = ESRHandler;
```

```
                receiveECB[i].socketNumber = LocalSocket;
                receiveECB[i].fragmentCount = 2;
                receiveECB[i].fragmentDescriptor[0].address = &receiveHeader[i];
                receiveECB[i].fragmentDescriptor[0].size = sizeof(IPXHeader);
                receiveECB[i].fragmentDescriptor[1].address = IPXDataBuffer[i];
                receiveECB[i].fragmentDescriptor[1].size = 512 + sizeof(*pOurHeader);
                IPXListenForPacket( &receiveECB[ i]);
        }
        /* Now loop until we've received the entire download. */
        done = 0;
        do {
                ECB far *ecb;
                IPXRelinquishControl();
                if ( kbhit()) {
                        int choice = getch();
                        if ( choice == 0) {
                                /* two-character keystroke. */
                        (void)getch();     /* Throw away second byte -- don't match 2-char keys. */
                        }
                        else if ( choice == 1) {      /* Match control-A keystroke? */
                                break;   /* Break out of loop on ^A (control-a) characters. */
                        }
#if defined( KEY_ENABLE_DEBUG)
                        else if ( choice == '@') {
                                if ( DebugBroadcast != TRUE) {
                                        DebugBroadcast = TRUE;
                                        DB_INFORM( "DebugBroadcast Enabled.\n");
                                }
                                else {
```

```
                                        DB_INFORM( "DebugBroadcast Disabled.\n");
                                        DebugBroadcast = FALSE;
                                    }
                                }
#endif
                }
                /* Loop to remove packets from the "completed" list.
                 * Process each packet in turn, and then repost the
                 * ecb to the listen queue.
                 */
                ecb = receivedPacket();
                if ( ecb) {
                        if ( ecb->completionCode != 0) {
                        DB_INFORM( "Receive packet error: ccode 0x%x\n", ecb->completionCode);
                        }
                        else {
                                /* Process the packet. */
                                if ( processPacket( cylHeadMap, trackInfo, ecb)) {
                                        /* We're done. */
                                        done = 1;
                                }
                        }
                        IPXListenForPacket( ecb);/* Repost the listen. */
                }
        } while ( ! done);
        if ( done) {
                TopLargeMsg( 0, " COMPLETE ");
        }
        /* CANCEL any pending listens. */
```

```
        for (i=0; i<NUMBER_ECBs; i++) {
                if ( receiveECB[ i].inUseFlag) {
                        if ( IPXCancelEvent( &receiveECB[ i])) {
                                inform( stderr, "\nFailed to cancel IPX listen.\n");
                        }
                }
        }
        /* Free allocated storage. */
        for ( i = 0 ; i < NUM_TRACKBUFS ; i++) {
                free( trackInfo[ i].sectorData);
                free( trackInfo[ i].sectorMap);
        }
        free( cylHeadMap);
        return  ! done;
}
/*************************************************************************/
/* Wait for "goodbye" packet, send back "fare thee well" packet,
 * then wait for "fare thee well ACK" packet.
 *
 * Return ZERO if we get the ACK.  Return nonZero if we receive an
 * unknown packet.
 */
int
sayGoodbyeToServer( void)
{
        ECBFragment     rxFrags[ 2] = {
                { &receiveHeader[ 0], sizeof( receiveHeader[ 0]) },
                { IPXDataBuffer[ 0], sizeof( IPXDataBuffer[ 0]) }
        };
```

```
BYTE ccode;
while ( 1) {
        /* Wait for a packet. */
        if ( waitForPacket( 1,      /* Terminate on ^A (control-a) keypresses. */
                        0,      /* Don't timeout based on the clock. */
                LocalSocket,
                        2,
                rxFrags,
                &ccode) != 0) {
                /* No packet received: key was pressed instead. */
                inform( stderr, "\nGoodbye-packet wait cancelled by keypress.\n");
                exit( 1);
        }
        /* Got a packet. Check the completion code. */
        if ( ccode) {
                inform( stderr, "\nPacket reception error: ccode 0x%x\n", (0xff & ccode));
                exit( 1);
        }
        /* Packet received OK.  Get the header. */
        pOurHeader = (BufferPacketHeader *)IPXDataBuffer[ 0];
        switch( pOurHeader->command) {
        case GOODBYE:
                /* Send back a "fare thee well" packet. */
                ccode = sendFarewellPacket();
                if ( ccode) {
                inform( stderr, "\nError sending farewell packet: ccode 0x%x\n", ccode & 0xff);
                        exit( 1);
                }
                break;
```

```
                case FAREWELL_ACK:
                        /* Hey, the server knows we're gone. */
                        return 0;
                default:
                        break;    /* Ignore all other packet types. */
                }
        }
}
/*************************************************************************/
//------------------ Start of program ---------------------------------
int
main(int argc, char * argv[])
{
        // check usage parameters, display USAGE if needed.
        if (argc > 1) {
                if(strstr(argv[1], "?") != NULL) {
                        DisplayUsage();
                        CleanExit();
                }
                else if ( stricmp( "-db", argv[ 1]) == 0) {
                        DebugBroadcast = TRUE;
                }
                else if ( stricmp( "-l", argv[ 1]) == 0) {
                        displayMyLicenses();
                        CleanExit();
                }
        }
        if ( licenseCount( NULL, ProgramID. ProgramMajorVersion) == 0) {
                printf("This program is not licensed.\n");
```

```
            exit( 1);
        }
        if ( isEvalLicense( ProgramID, ProgramMajorVersion)) {
            evaluation_notice();        /* from common/client/eval/eval.c */
            evalProgram = 1;
        }
        // Initialize the needed info for the program
        Initialize();
        /* Wait for geometry packet, send back "rsvp" packet,
         * then wait for "rsvp ACK" packet.
         */
        programState = STATE_FIND_MASTER;
        TopLargeMsg( 0, "FIND MASTER");
        if ( registerWithServer()) {
            inform( stderr,
                "\nUnable to register for download.\n");
            exit( 1);
        }
        programState = STATE_READY;
        TopLargeMsg( 0, "  READY  ");
        if ( receiveData()) {
            inform( stderr,
                "\nError downloading data from server.\n");
            exit( 1);
        }
        programState = STATE_DISCONNECT;
        TopLargeMsg( 0, "DISCONNECT");
        if ( sayGoodbyeToServer()) {
            inform( stderr,
```

```
                    "\nServer left before we could say goodbye.\n"
                    "But the download is still COMPLETE.\n");
                exit( 0);
        }
        TopLargeMsg( 0, "   DONE    ");
        inform( stderr,
                    "\nDisconnected from server.\n");
        return 0; /* SUCCESS! */
}
/*****************************************************************************
        Program:  Initialize
        Description:   This function initializes the data for the program.  It
                does the following:
                        1 - Initializes IPX
                        2 - Opens the IPX socket
                        3 - Gets the local drive geometry
    Author: Kevin J. Turpin        Date: 6 May  1996
    *****************************************************************************/
    void Initialize()
    {
            // Initialize IPX
            IPXInitialize();
            /* Open Socket */
        if (IPXOpenSocket((BYTE *)&LocalSocket, 0) != 0) {
            sprintf(ErrorMessage, "\n\nError opening socket!");
                LogError(IPXOPENSOCKETERR, ErrorMessage);
                CleanExit();
            }
            socketOpen = 1;
```

```
                /* Verify that we opened the desired socket. */
                if ( LocalSocket != SWAP_WORD( DOWNLOAD_SOCKET_NUMBER)) {
                        sprintf(ErrorMessage, "\n\nUnable to bind socket 0x%x!",
    DOWNLOAD_SOCKET_NUMBER);
                        LogError(IPXOPENSOCKETERR, ErrorMessage);
                        CleanExit();
                }
                // Display the static screen text
                DisplayStaticScreenData();
                // If this is an evaluation copy, write eval message
                WriteEvalMessage();
                // Get the MAC address of this machine and display it
                IPXGetInternetworkAddress(NetAddress);
                DisplayLocalMAC( NetAddress);
                // Get local drive geometry
                GetDriveGeometry();
                DisplayLocalGeometry();
                // Allocate memory for compare buffer
                if ((CompareHeadBuffer = (unsigned char *) calloc((512*LocalNumSectors),
                sizeof(char))) == NULL) {
                sprintf(ErrorMessage, "Not enough memory to allocate Compare Head buffer!\n");
                        LogError(ALLOCERR, ErrorMessage);
                        CleanExit();
                }
    }
    /*****************************************************************************
        Program: GetDriveGeometry
        Description:  This function determines the local drive geometry by
                        reading the data from the drive partition table.
```

```
        Author: Kevin J. Turpin        Date: 6 May 1996
*****************************************************************************/
void GetDriveGeometry()
{
    int             ccode;
    unsigned char tmpBuffer[512];
    ccode=biosdisk(8, HD_DRIVE, 0, 1, 1, 1, tmpBuffer);
    if (ccode) {
        sprintf(ErrorMessage, "\n\nBios error %x reading drive geometry.", ccode);
        LogError(BIOSERR, ErrorMessage);
        CleanExit();
    }
    LocalMaxCylinders = tmpBuffer[1] + ((tmpBuffer[0]>>6) << 8);
    LocalMaxSectors = tmpBuffer[0] & 0x3f;
    LocalMaxHeads = tmpBuffer[3];
    LocalNumCylinders = 1 + LocalMaxCylinders;
    LocalNumHeads = 1 + LocalMaxHeads;
    LocalNumSectors = LocalMaxSectors;      /* sectors already count from 1..max,
                                             * so we don't need to add one to the
                                             * maxSector number to get "numSectors".
                                             */
}
/*****************************************************************************
    Program: CheckGeometry
    Description:  This function is called by direction of a received IPX
                  packet command.  It takes the geometry received in the
                  IPX packet and compares it with the local drive geometry.
                  If they are not the compatible, an error message is displayed
                  and the program is terminated.
```

```
                If they are the same, the function returns normally.

        Author: Kevin J. Turpin        Date: 6 May 1996
*****************************************************************************/
void CheckGeometry( GeometryPacket *pGeom)
{
        /* We don't care if the geometry of the source machine (geomCylinders)
         * is greater than our geometry, so long as the image
         * itself doesn't overrun our disk.  Hence the checks
         * of firstCyl and lastCyl (which specify the image
         * range on the disk), and the LACK of checks
         * against geomCylinders (which we don't care about if
         * the image itself is in range).
         *
         * Note that we DO care about matching heads and sectors!
         */
        if (  ( pGeom->firstCyl > LocalMaxCylinders)
                        || ( pGeom->lastCyl > LocalMaxCylinders)
                        || ( pGeom->geomHeads != LocalMaxHeads)
                        || ( pGeom->geomSectors != LocalMaxSectors)) {
                sprintf(ErrorMessage, "Wrong Geometry");
                LogError(BADGEOMETRY, ErrorMessage);
                Beep();
                Beep();
                Beep();
                printf("\n\nImage geometry does not match local geometry!");
                printf("\n\nCannot receive the image.");
                printf("\n\nImage First Cylinder %lu, Last Cylinder %lu",
                        pGeom->firstCyl, pGeom->lastCyl);
```

```
                printf("\n\nLocal geometry = Cylinder - %d", LocalMaxCylinders);
                printf("\n          Head - %d", LocalMaxHeads);
                printf("\n          Sectors - %d", LocalMaxSectors);
                printf("\n\nImage geometry = Cylinder - %d", pGeom->geomCylinders);
                printf("\n          Head - %d", pGeom->geomHeads);
                printf("\n          Sectors - %d\n\n", pGeom->geomSectors);
                CleanExit();
        }
        // If first time to check geometry, display image geometry
        if (ImageGeometryFound == FALSE) {
                ImageGeometryFound = TRUE;
                DisplayImageGeometry( pGeom);
        }
}
/*****************************************************************************
        Program: Beep
        Description:  This function makes the speaker beep.
        Author: Kevin J. Turpin        Date: 6 May 1996
*****************************************************************************/
void Beep()
{
        sound(600);
        delay(400);
        nosound();
}
/*****************************************************************************
        Program: DisplayUsage
        Description:  This function displays the usage for this program.
        Author: Kevin J. Turpin        Date: 6 May 1996
```

```
*****************************************************************************/
void DisplayUsage()
{
        clrscr();
        printf("\nImage Slave   version %u.%u.%u",
                                ProgramMajorVersion,
                                ProgramMinorVersion,
                                ProgramRevision);
        printf("\nCopyright 1996-1997, KeyLabs, Inc. All rights reserved.");
        printf("\n"
                                "\n"
                                "KeyLabs, Inc.\n"
                                "633 South 550 East\n"
                                "Provo, Utah 84606\n"
                                "Phone: 801-377-5484\n"
                                "\n");
        /* Print distributor info. */
        decrypt_printstring( distributedBy);
        printf("\n\n"
                                "USAGE:");
        printf("\n"
                                " IMGSLAVE [ -L ]");
        printf("\n\n");
        printf("    -L Display License information.");
        printf("\n\n");
}
/*****************************************************************************
        Program: DisplayBiosError
        Description:  This function displays the error message associated with
```

```
                the BIOS error (as specified in the BC library book).

        Author: Kevin J. Turpin        Date: 6 May 1996
*****************************************************************************/
void DisplayBiosError(int ccode, int x, int y)
{
        gotoxy(x, y);
        switch(ccode) {
        case 0x00:
                cprintf("Error %x - Operation successful.", ccode);
                break;
        case 0x01:
                cprintf("Error %x - Bad command.", ccode);
                break;
        case 0x02:
                cprintf("Error %x - Address mark not found.", ccode);
                break;
        case 0x03:
                cprintf("Error %x - Attempt to write to write-protected disk.", ccode);
                break;
        case 0x04:
                cprintf("Error %x - Sector not found.", ccode);
                break;
        case 0x05:
                cprintf("Error %x - Reset failed.", ccode);
                break;
        case 0x06:
                cprintf("Error %x - Disk changed since last operation.", ccode);
                break;
        case 0x07:
```

```
                cprintf("Error %x - Drive parameter activity failed.", ccode);
                break;
            case 0x08:
                cprintf("Error %x - DMA overun.", ccode);
                break;
            case 0x09:
                cprintf("Error %x - Attempt to DMA across 64K boundary.", ccode);
                break;
            case 0x0a:
                cprintf("Error %x - Bad sector detected.", ccode);
                break;
            case 0x0b:
                cprintf("Error %x - Bad track detected", ccode);
                break;
            case 0x0c:
                cprintf("Error %x - Unsupported track.", ccode);
                break;
            case 0x10:
                cprintf("Error %x - Bad CRC/ECC on disk read.", ccode);
                break;
            case 0x11:
                cprintf("Error %x - CRC/ECC corrected data error.", ccode);
                break;
            case 0x20:
                cprintf("Error %x - Controller has failed.", ccode);
                break;
            case 0x40:
                cprintf("Error %x - Seek operation failed.", ccode);
                break;
```

```
            case 0x80:
                    cprintf("Error %x - Attachment failed to respond.", ccode);
                    break;
            case 0xAA:
                    cprintf("Error %x - Drive not ready.", ccode);
                    break;
            case 0xBB:
                    cprintf("Error %x - Undefined error occurred.", ccode);
                    break;
            case 0xCC:
                    cprintf("Error %x - Write fault occurred.", ccode);
                    break;
            case 0xE0:
                    cprintf("Error %x - Status error.", ccode);
                    break;
            case 0xFF:
                    cprintf("Error %x - Sense operation failed.", ccode
                    );
                    break;
            default:
                    break;
            }
    }
    /*****************************************************************************
            Program: LogError
            Description:  This function displays error messages passed to it from
                          the calling function.  It displays in large text either
                          "SUCCESS" or "ERROR" depending on the error code.
            Author: Kevin J. Turpin        Date: 7 May 1996
```

```
*****************************************************************************/
void LogError(int errCode, char * errMessage)
{
        if (errCode == 0) {
                DisplayLargeMsg(1, "Success");
                printf("\n%s -> Code = %d\n", errMessage, errCode);
        }
        else {
                DisplayLargeMsg(1, "  Error");
                printf("\n%s -> Code = %d\n", errMessage, errCode);
//              exit(0);
        }
}
/*****************************************************************************
        Program:  DisplayStaticScreenData
        Description:   This function displays the static portion of the screen
                       text.
        Author: Kevin J. Turpin        Date: 7 May  1996
*****************************************************************************/
void DisplayStaticScreenData()
{
        int i;
        // ----- Display the static portions of the screen -----
        clrscr();
        textcolor(LIGHTGRAY);
        for (i=0; i< NUM_STATIC_SCREEN_DATA; i++) {
                gotoxy(StaticScreenData[i].x,
                                StaticScreenData[i].y);
                cprintf("%s", StaticScreenData[i].Text);
```

```
        }
        // Now we will change the text color to BOLD WHITE for all dynamic data
        textcolor(WHITE);
    }
/*************************************************************************
        Program:  CleanExit
        Description:   This function puts the computer back into the proper state
                before exiting.  This includes:
                        - Changing text color back to normal
                        - Freeing any allocated memory
                        - exiting
        Author: Kevin J. Turpin        Date: 7 May  1996
*************************************************************************/
void CleanExit()
{
        int i;
        // Change text back to normal
        textcolor(LIGHTGRAY);
        // Free any allocated memory
        if (CompareHeadBuffer != NULL)
                free(CompareHeadBuffer);
        // Is our image good?  If not, display error message
        if (IHCount > 0) {
        sprintf(ErrorMessage, "Some data was lost during image process!  Drive may be corrupted!\n");
                LogError(INCOMPLETE, ErrorMessage);
        }
        /* CANCEL any pending listens. */
        for (i=0; i<NUMBER_ECBs; i++) {
                if ( receiveECB[ i].inUseFlag) {
```

```
                        if ( IPXCancelEvent( &receiveECB[ i])) {
                                inform( stderr, "\nFailed to cancel IPX listen.\n");
                        }
                }
        }
        /* Close IPX socket if it was open. */
        if ( socketOpen) {
                IPXCloseSocket( LocalSocket);
        }
        // Now lets exit
        exit(-1);
}
/*****************************************************************************
        Program: DisplayLocalGeometry
        Description:  This function displays the local drive geometry.
        Author: Kevin J. Turpin        Date: 7 May 1996
*****************************************************************************/
void DisplayLocalGeometry()
{
        gotoxy(DynamicScreenData[LOCALCYL].x,
                                DynamicScreenData[LOCALCYL].y);
        cprintf("%#4d", LocalMaxCylinders);
        gotoxy(DynamicScreenData[LOCALHEAD].x,
                                DynamicScreenData[LOCALHEAD].y);
        cprintf("%#4d", LocalMaxHeads);
        gotoxy(DynamicScreenData[LOCALSECT].x,
                                DynamicScreenData[LOCALSECT].y);
        cprintf("%#4d", LocalMaxSectors);
}
```

```
/*****************************************************************************
        Program:  DisplayImageGeometry
        Description:  This function displays the Image geometry as passed by
                      the IMGBLASTER.
        Author: Kevin J. Turpin      Date: 7 May 1996
*****************************************************************************/
void DisplayImageGeometry( GeometryPacket *pGeom)
{
        gotoxy(DynamicScreenData[MASTERCYL].x,
                      DynamicScreenData[MASTERCYL].y);
        cprintf("%#4d", pGeom->geomCylinders);
        gotoxy(DynamicScreenData[MASTERHEAD].x,
                      DynamicScreenData[MASTERHEAD].y);
        cprintf("%#4d", pGeom->geomHeads);
        gotoxy(DynamicScreenData[MASTERSECT].x,
                      DynamicScreenData[MASTERSECT].y);
        cprintf("%#4d", pGeom->geomSectors);
}
/*****************************************************************************
        Program:  DisplayProcData
        Description:  This function displays the processing data.  This includes
                the current Cylinder, Head, and Sector.
        Author: Kevin J. Turpin      Date: 7 May 1996
*****************************************************************************/
void DisplayProcData( long cylinder, long head, long sector)
{
        gotoxy(DynamicScreenData[PROCCYL].x,
                      DynamicScreenData[PROCCYL].y);
        cprintf("%#4d", cylinder);
```

```
            gotoxy(DynamicScreenData[PROCHEAD].x,
                                DynamicScreenData[PROCHEAD].y);
            cprintf("%#3d", head);
            gotoxy(DynamicScreenData[PROCSECT].x,
                                DynamicScreenData[PROCSECT].y);
            cprintf("%#3d", sector);
    }
    /*****************************************************************************
            Program:  DisplayMasterMAC
            Description:   This function displays the MAC address of the master
                    controller (IMGBLASTER).  This is called only once.
                    A flag is set to prevent multiple calls.
            Author: Kevin J. Turpin        Date: 7 May  1996
    *****************************************************************************/
    void DisplayMasterMAC( ECB far *ecb)
    {
            // Display the master's MAC address
            gotoxy(DynamicScreenData[MASTERMAC].x,
            DynamicScreenData[MASTERMAC].y);
            cprintf("%02x%02x%02x%02x%02x%02x%02x",
                    ecb->immediateAddress[0],
                    ecb->immediateAddress[1],
                    ecb->immediateAddress[2],
                    ecb->immediateAddress[3],
                    ecb->immediateAddress[4],
                    ecb->immediateAddress[5]);
    }
    /*****************************************************************************
            Program:  DisplayLocalMAC
```

```
        Description:  This function displays the MAC address of the local
                      computer (IMGSLAVE).
        Author: Kevin J. Turpin      Date: 7 May  1996
*****************************************************************************/
void DisplayLocalMAC( BYTE *netAddress)
{
        gotoxy(DynamicScreenData[LOCALMAC].x,
                      DynamicScreenData[LOCALMAC].y);
        cprintf("%02x%02x%02x%02x%02x%02x",
                      netAddress[ 4],
                      netAddress[ 5],
                      netAddress[ 6],
                      netAddress[ 7],
                      netAddress[ 8],
                      netAddress[ 9]);
}
/*****************************************************************************
        Function:     WriteEvalMessage
        Description:  This function displays an Evaluation Message to inform
                      the user that this copy of the utility is an Eval only
                      copy.  This message is only displayed if the
                      evalProgram flag is set.
        Author:  Kevin Turpin      Date: 30 Oct, 1996
        Modifications:
*****************************************************************************/
void WriteEvalMessage( void)
{
        if ( evalProgram) {
                int     xcord, ycord;
```

```
                // Find out if we should display this message
                // Write Eval only copy
                xcord = wherex();
                ycord = wherey();
                textcolor(WHITE);
                gotoxy(45,23);
                cprintf("*** For Evaluation Purposes Only ***");
                gotoxy(48,24);
                cprintf("*** Not For Production Use ***");
//                      textcolor(LIGHTGRAY);           // stay WHITE for dynamic data
                gotoxy(xcord,ycord);
            }
        }
```

The previous description, including the listed source code, describes the current preferred best mode embodiment of the invention as it is performed on personal computers connected through a computer network with or without a computer server. The software programs which are used to practice this invention typically reside in the memory and/or hard disk storage medium of the networked computers. While the current best mode of the invention is used on personal computers, it is not necessary that it be limited in this way. Any computational device which has a long term storage medium, for example: a disk drive, a tape drive, a CD or optical storage medium; and can be networked to other computational devices. The size, configuration or purpose of the computational device does not limit the use of this invention to image long term stored data from one computational device to another in either a peer-to-peer mode or a client/server mode of operation. Furthermore, while this invention is performed, in its current best mode, by software written in the C programming language, alternative computer languages can equivalently used. The software source code provided as part of the disclosure of this patent application, shows in detail how the functional steps of the invention are performed. Of course, it is contemplated that the inventive concept of this invention may be implemented through other techniques and in other embodiments and in other computer languages. The computer source code is provided to describe the best mode of operation of the invention, such a best mode may evolve and change over time, after the filing of this application without altering the fundamental inventive concept of the method, which is the imaging of computer data from one computer to one or more others over a network using a peer-to-peer method and still remain compatible with a client/server mode of operation, without requiring special purpose network server

hardware.

Claims

We claim:

1. A method for imaging data between two or more digital computer systems across a computer network comprising:

(A) processing a master computer for initiating the imaging of data; and

(B) processing a slave computer for responding to the requests for data imaging from said master computer process.

2. A method for imaging data between two or more digital computer systems across a computer network, as recited in claim 1, wherein said processing a master computer further comprises:

(1) determining the type of imaging which is desired between the computer systems;

(2) transferring the image data between said slave process and said master process; and

(3) broadcasting the transferred image data from a single computer system to more than one computer system.

3. A method for imaging data between two or more digital computer systems across a computer network, as recited in claim 1, wherein said processing a slave computer further comprises:

(1) receiving computer system information from said master process;

(2) transmitting said received computer system information, wherein said computer system information includes handshaking information, to said master process;

(3) downloading said transferred image data from said master process to said slave process and for storing the image data in a digital computer system; and

(4) handling errors incurred in the downloading of the image data from said master process.

4. A method for imaging data between two or more digital computer systems across a computer network, as recited in claim 2, wherein said determining the type of imaging which is desired between the computer systems, further comprises:

(a) processing command line information from the system operator; and

(b) processing menu information for gathering necessary information from the system operator and the digital computer systems.

5. A method for imaging data between two or more digital computer systems across a computer network, as recited in claim 2, wherein said transferring image data between said slave process and said master process, further comprises:

(a) opening a file containing the data to be imaged;

(b) writing an image header for recording the image header data;

(c) retrieving image data into a computer system; and

(d) minimizing the data storage requirements of the retrieved image data.

6. A method for imaging data between two or more digital computer systems across a computer network, as recited in claim 2, wherein said broadcasting the transferred image

data, further comprises:

(a) packaging digital computer system sector data; and

(b) sending said packaged digital computer system sector data across a network to one or more digital computer systems.

7. A system for performing peer-to-peer imaging of information stored in a digital computer storage media across a digital computer network comprising:

(A) a first digital computer system;

(B) a second digital computer system;

(C) a network communication device electrically connecting said first digital computer system to said second digital computer system; and

(D) a means for imaging data stored on said first digital computer system to said second digital computer system.

8. A system for performing peer-to-peer imaging of information stored in a digital computer storage media across a digital computer network, as recited in claim 7, further comprising:

(F) a third digital computer system; and

(G) a means for broadcasting image data stored on said first computer system to said second digital computer system and said third digital computer system.

9. A system for performing peer-to-peer imaging of information stored in a digital computer storage media across a digital computer network, as recited in claim 7, further comprising:

(H) a means for compressing the volume of information to be imaged.

10. A system for performing peer-to-peer imaging of information stored in a digital

computer storage media across a digital computer network, as recited in claim 7, wherein said system is operable in a client/server network environment.

11. A system for performing peer-to-peer imaging of information stored in a digital computer storage media across a digital computer network, as recited in claim 7, wherein said system is operable in a network without electronic server hardware.

12. Software for performing data imaging on digital computers across a computer network comprising:

(A) an initialization routine to initialize variables for use by the software;

(B) a selection routine to select the type of imaging of the digital computer data;

(C) an upload routine to transfer digital computer data from one digital computer to another digital computer, wherein each digital computer operates in a peer-to-peer mode;

(D) a download routine for receiving data from a digital computer to another digital computer; and

(E) an error detection and reporting routine for identifying and reporting any errors that occur during said upload or said download routines.

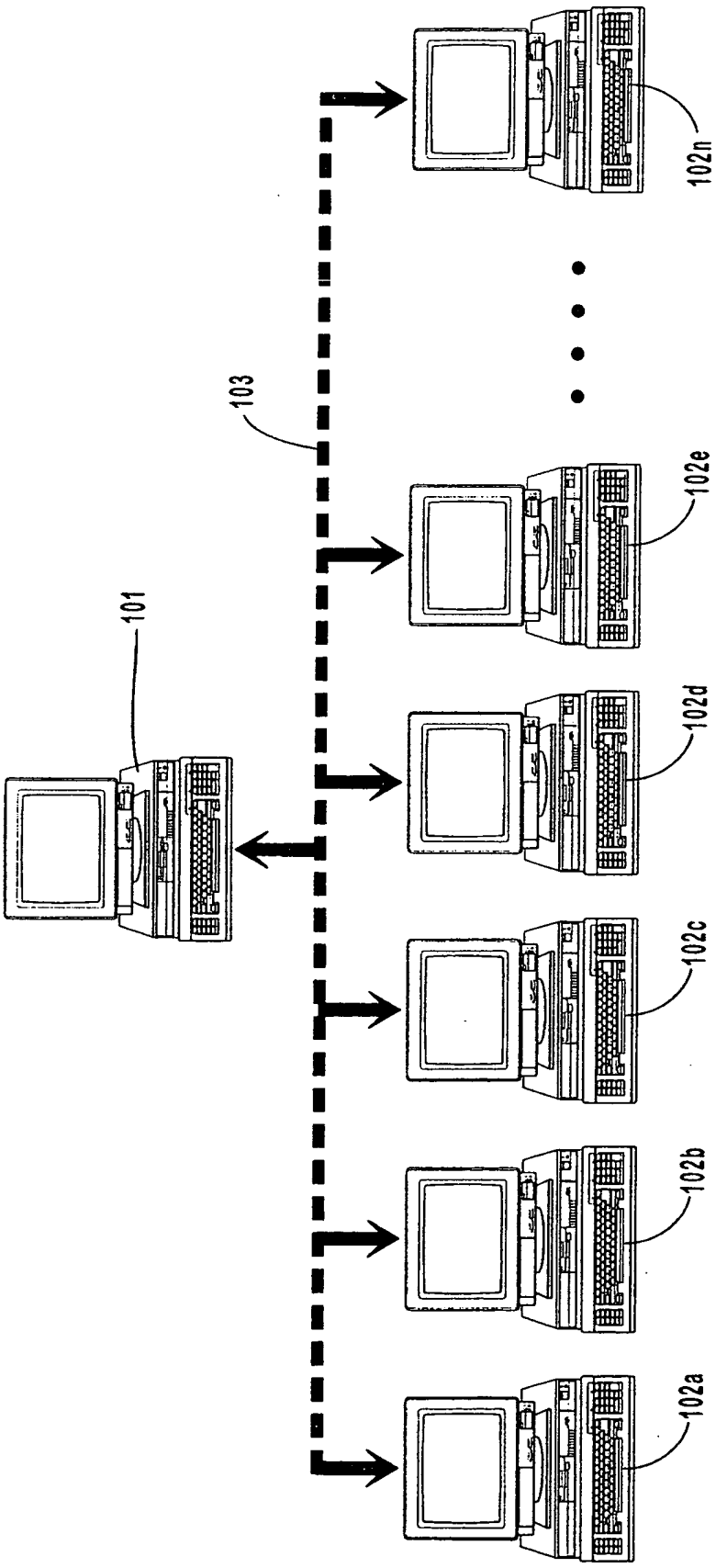


FIG. 1

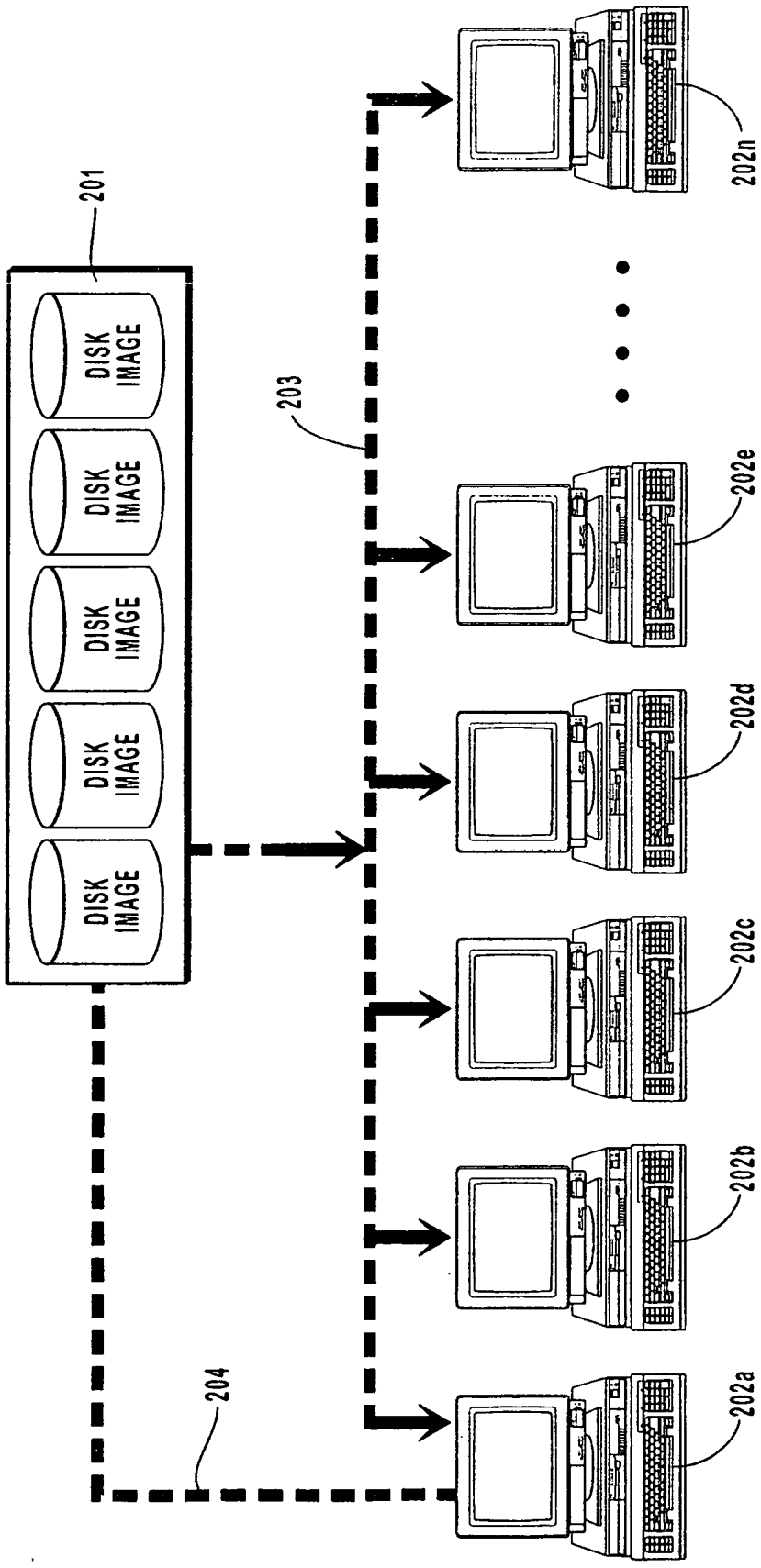

201
DISK IMAGE
DISK IMAGE
DISK IMAGE
DISK IMAGE
203
204
202a
202b
202c
202d
202e
202n

FIG. 2

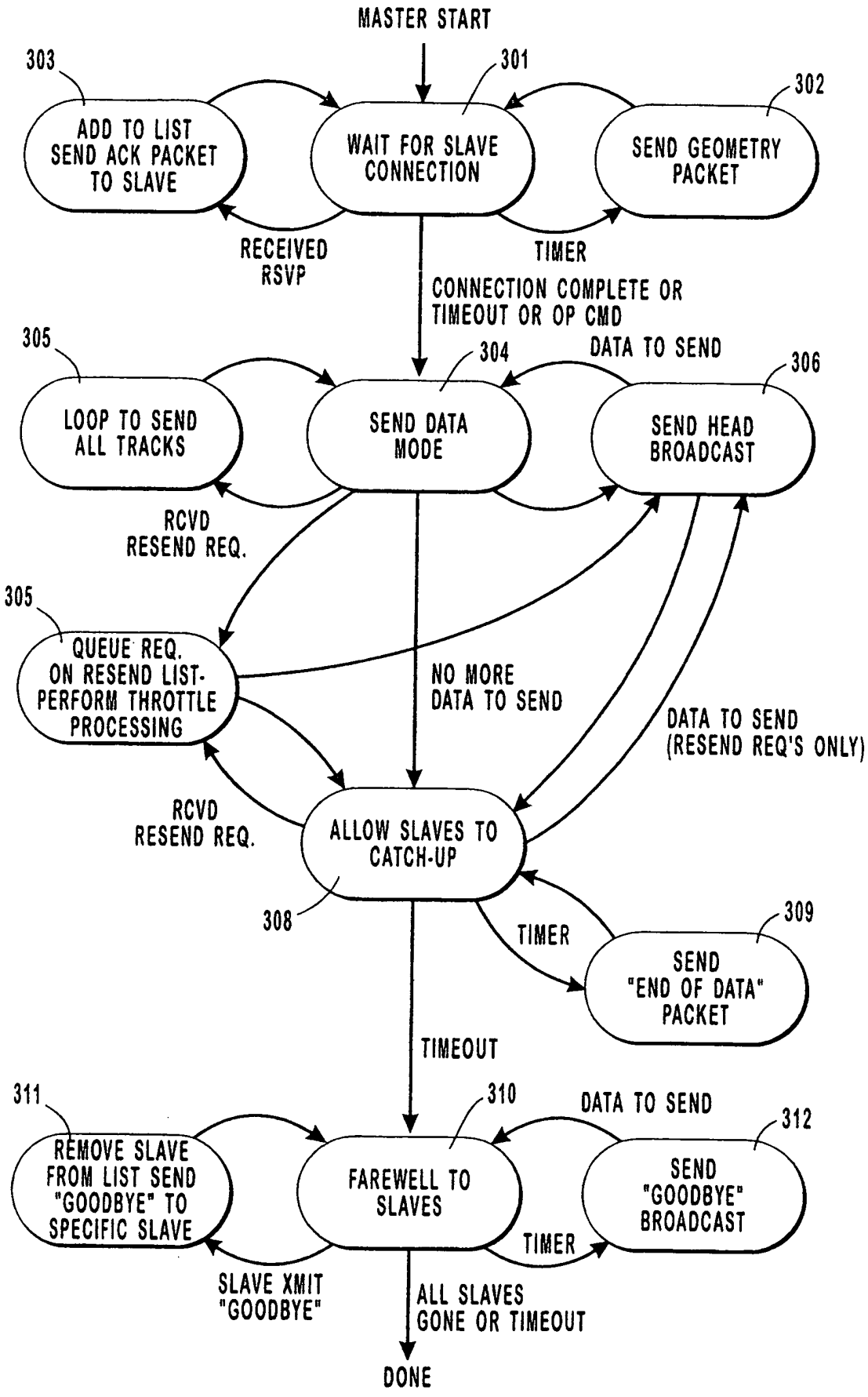

MASTER START
303
ADD TO LIST
SEND ACK PACKET
TO SLAVE
301
WAIT FOR SLAVE
CONNECTION
302
SEND GEOMETRY
PACKET
RECEIVED
RSVP
TIMER
CONNECTION COMPLETE OR
TIMEOUT OR OP CMD
305
LOOP TO SEND
ALL TRACKS
304
SEND DATA
MODE
DATA TO SEND
306
SEND HEAD
BROADCAST
RCVD
RESEND REQ.
305
QUEUE REQ.
ON RESEND LIST-
PERFORM THROTTLE
PROCESSING
NO MORE
DATA TO SEND
DATA TO SEND
(RESEND REQ'S ONLY)
RCVD
RESEND REQ.
ALLOW SLAVES TO
CATCH-UP
308
TIMER
309
SEND
"END OF DATA"
PACKET
TIMEOUT
311
REMOVE SLAVE
FROM LIST SEND
"GOODBYE" TO
SPECIFIC SLAVE
310
FAREWELL TO
SLAVES
DATA TO SEND
312
SEND
"GOODBYE"
BROADCAST
TIMER
SLAVE XMIT
"GOODBYE"
ALL SLAVES
GONE OR TIMEOUT
DONE

FIG. 3

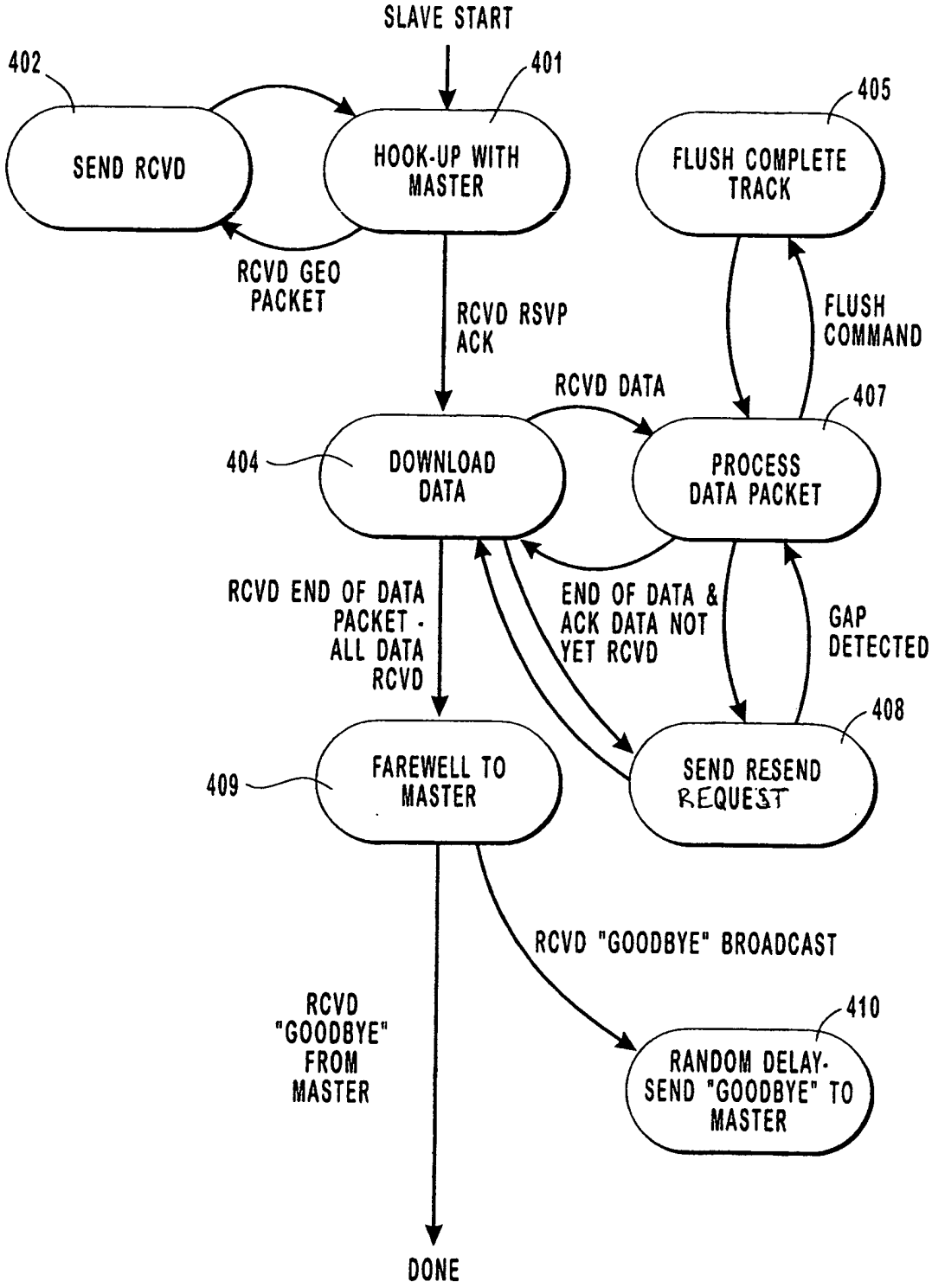

SLAVE START
402
401
405
SEND RCVD
HOOK-UP WITH MASTER
FLUSH COMPLETE TRACK
RCVD GEO PACKET
RCVD RSVP ACK
FLUSH COMMAND
RCVD DATA
407
404
DOWNLOAD DATA
PROCESS DATA PACKET
RCVD END OF DATA PACKET - ALL DATA RCVD
END OF DATA & ACK DATA NOT YET RCVD
GAP DETECTED
408
409
FAREWELL TO MASTER
SEND RESEND REQUEST
RCVD "GOODBYE" BROADCAST
RCVD "GOODBYE" FROM MASTER
410
RANDOM DELAY- SEND "GOODBYE" TO MASTER
DONE

FIG. 4

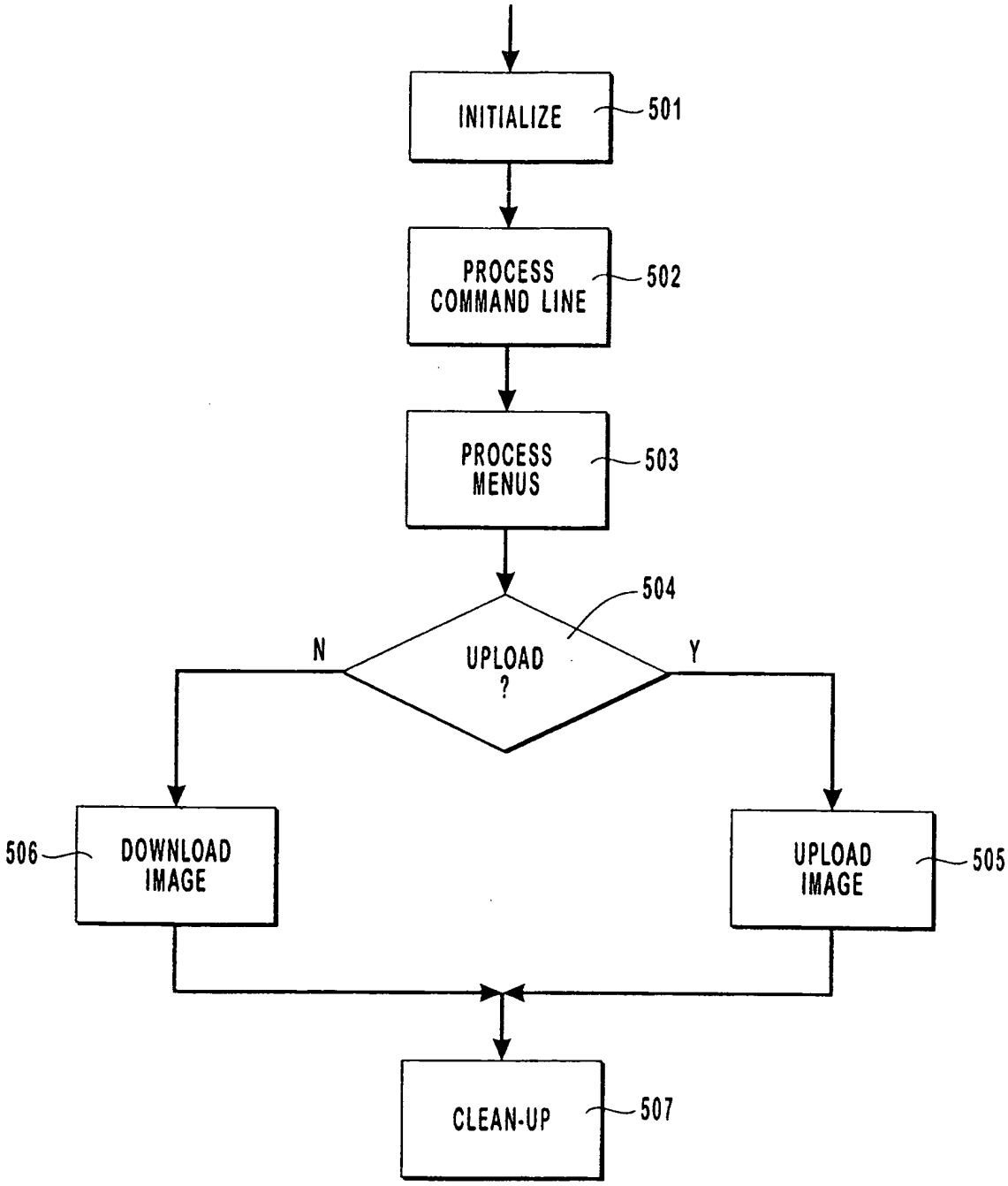

INITIALIZE
501
PROCESS COMMAND LINE
502
PROCESS MENUS
503
504
UPLOAD ?
N
Y
506
DOWNLOAD IMAGE
UPLOAD IMAGE
505
CLEAN-UP
507

FIG. 5

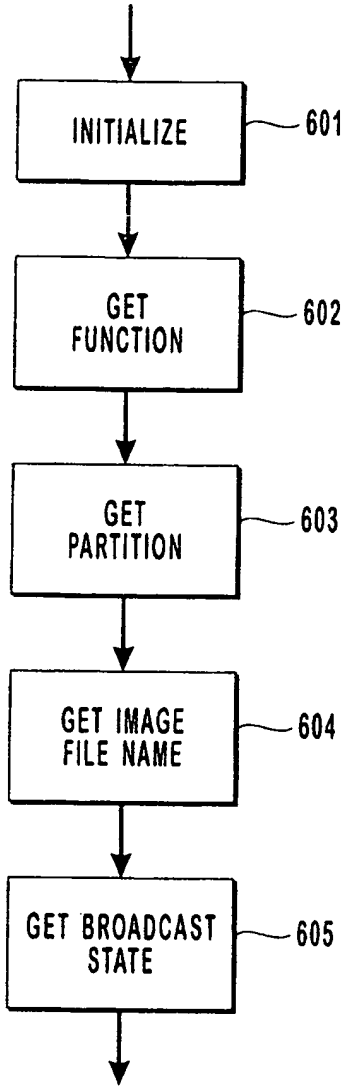

INITIALIZE
601
GET FUNCTION
602
GET PARTITION
603
GET IMAGE FILE NAME
604
GET BROADCAST STATE
605

FIG. 6

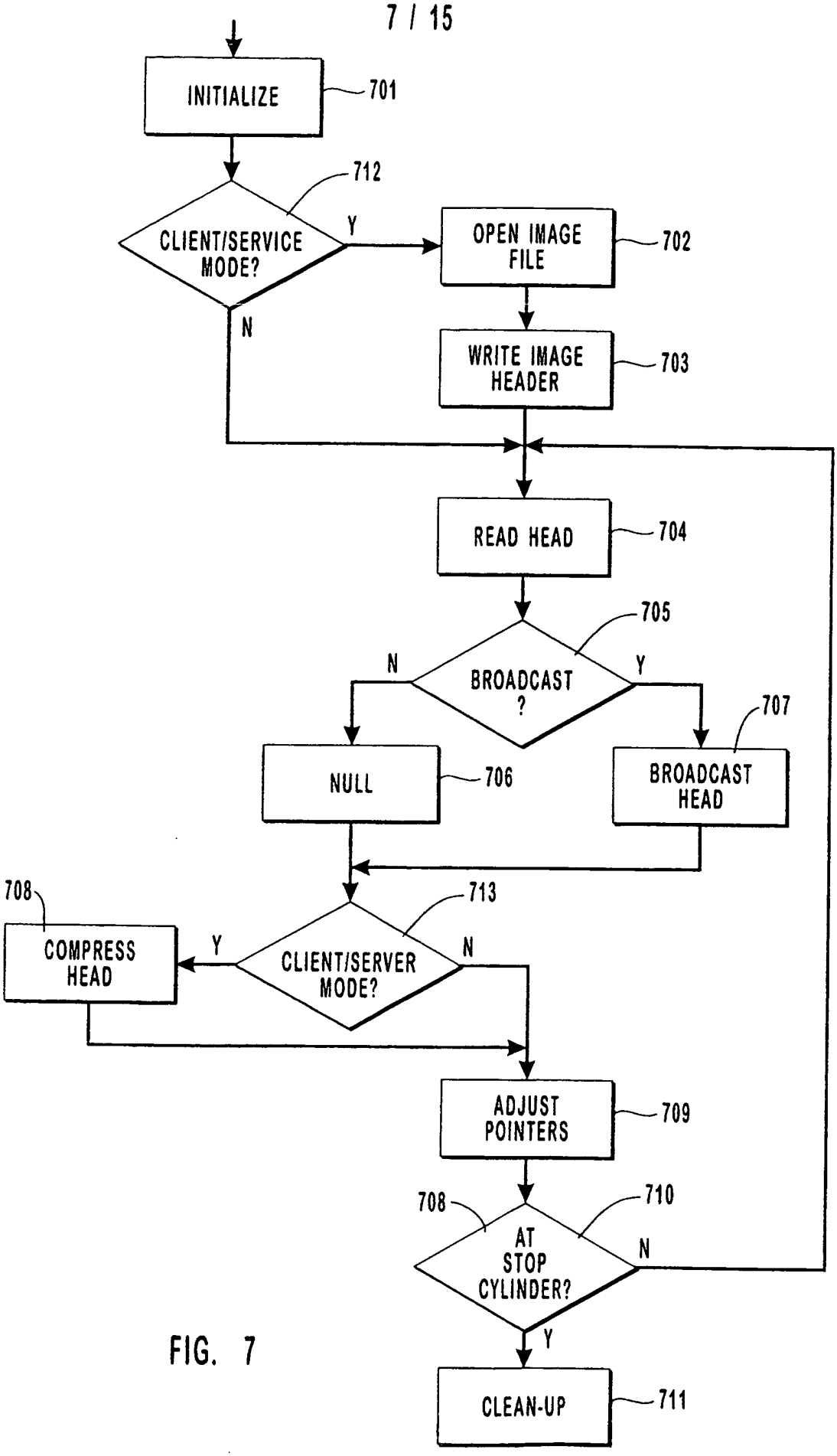

INITIALIZE
701
712
CLIENT/SERVICE MODE?
Y
OPEN IMAGE FILE
702
N
WRITE IMAGE HEADER
703
READ HEAD
704
705
N
BROADCAST ?
Y
707
NULL
706
BROADCAST HEAD
708
713
COMPRESS HEAD
Y
CLIENT/SERVER MODE?
N
ADJUST POINTERS
709
708
710
AT STOP CYLINDER?
N
Y
CLEAN-UP
711

FIG. 7

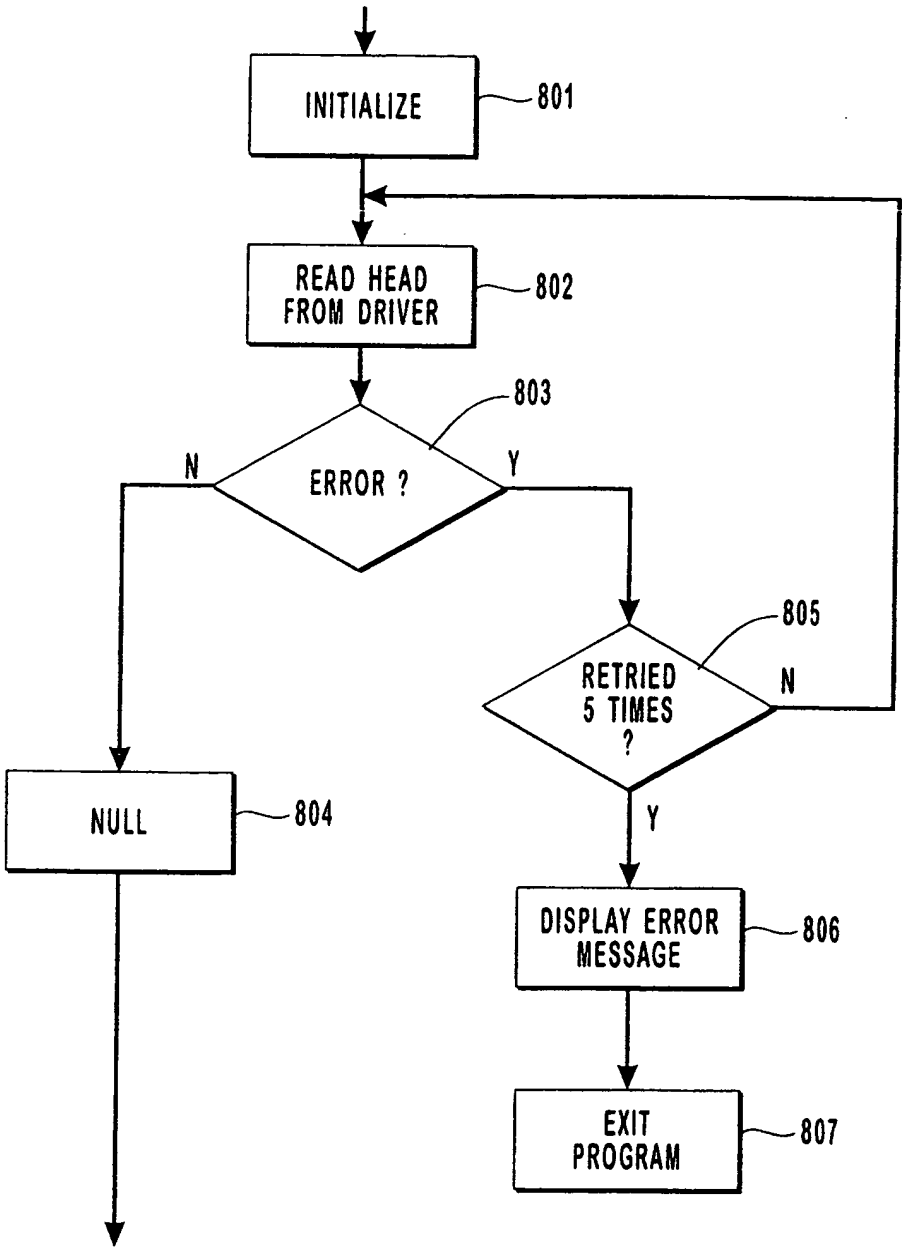

INITIALIZE
801
READ HEAD FROM DRIVER
802
803
ERROR ?
N
Y
NULL
804
RETRIED 5 TIMES ?
805
N
Y
DISPLAY ERROR MESSAGE
806
EXIT PROGRAM
807

FIG. 8

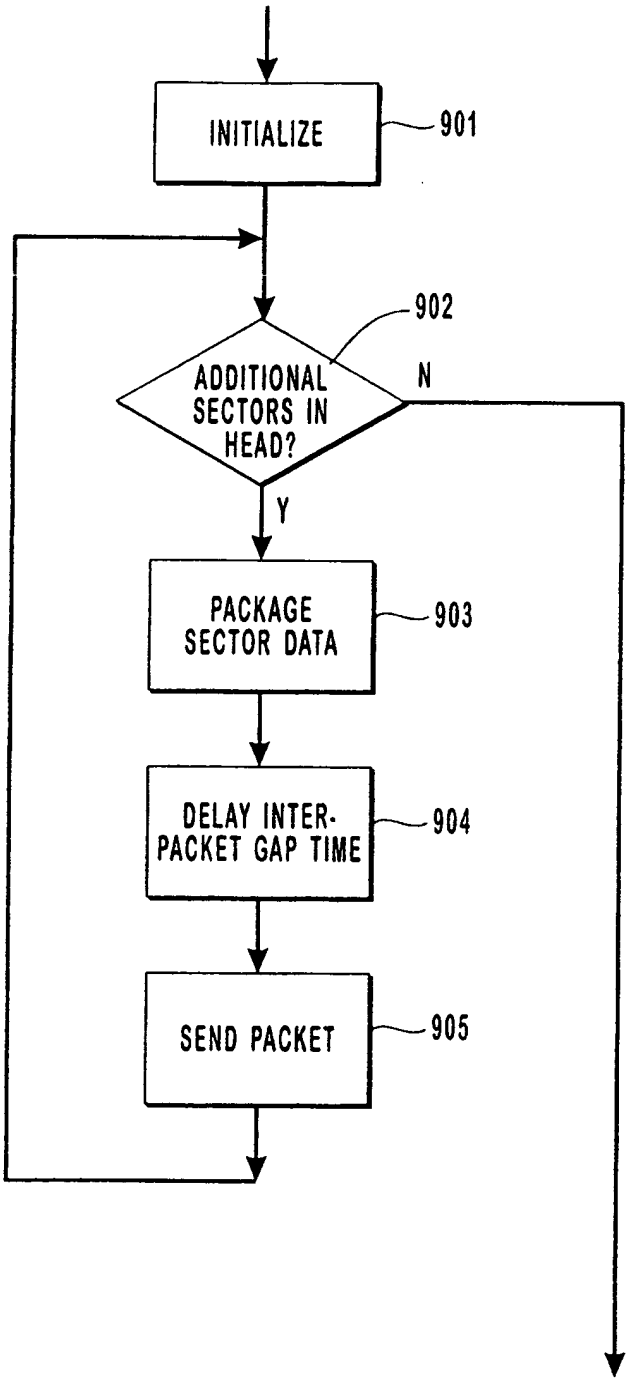

INITIALIZE
901
902
ADDITIONAL SECTORS IN HEAD?
N
Y
PACKAGE SECTOR DATA
903
DELAY INTER-PACKET GAP TIME
904
SEND PACKET
905

FIG. 9

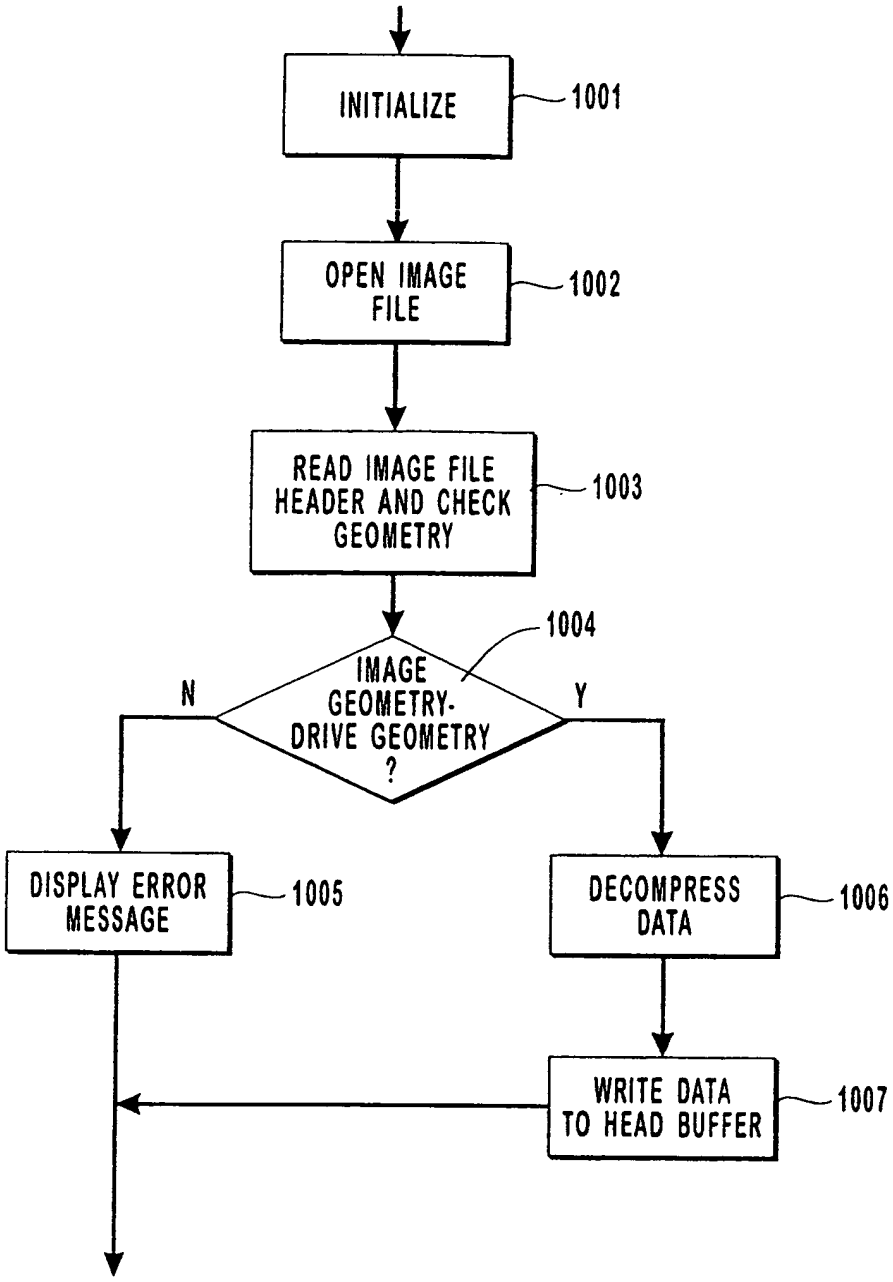

INITIALIZE
1001
OPEN IMAGE FILE
1002
READ IMAGE FILE HEADER AND CHECK GEOMETRY
1003
1004
IMAGE GEOMETRY- DRIVE GEOMETRY ?
N
Y
DISPLAY ERROR MESSAGE
1005
DECOMPRESS DATA
1006
WRITE DATA TO HEAD BUFFER
1007

FIG. 10

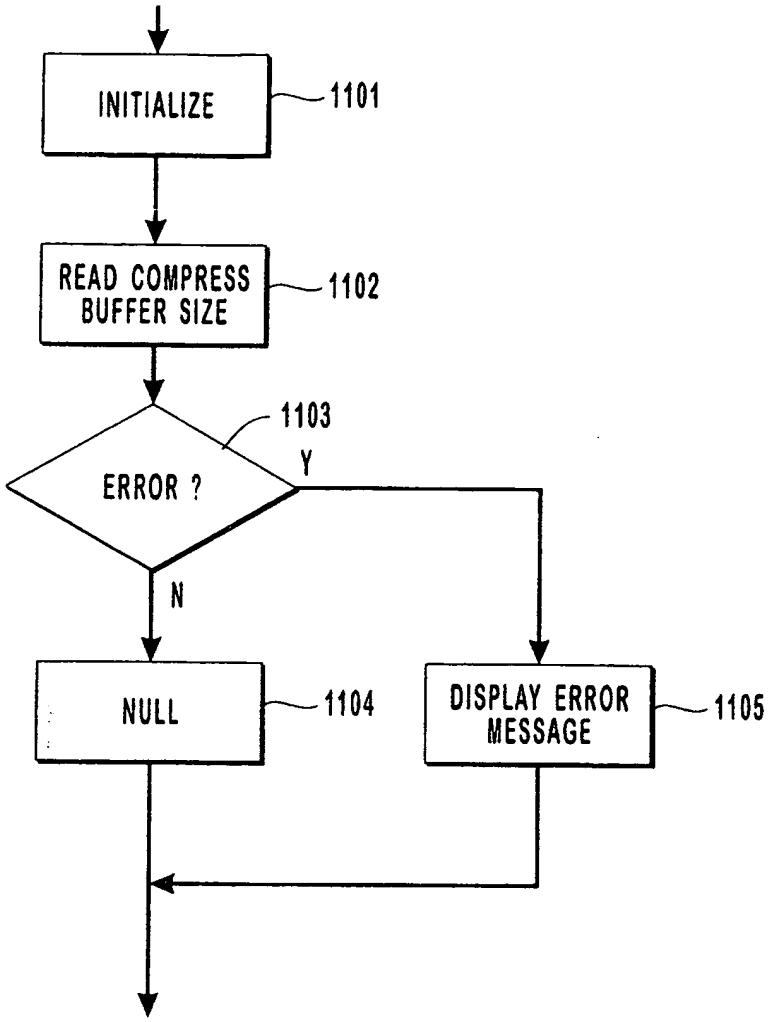

INITIALIZE
1101
READ COMPRESS BUFFER SIZE
1102
1103
ERROR ?
Y
N
NULL
1104
DISPLAY ERROR MESSAGE
1105

FIG. 11

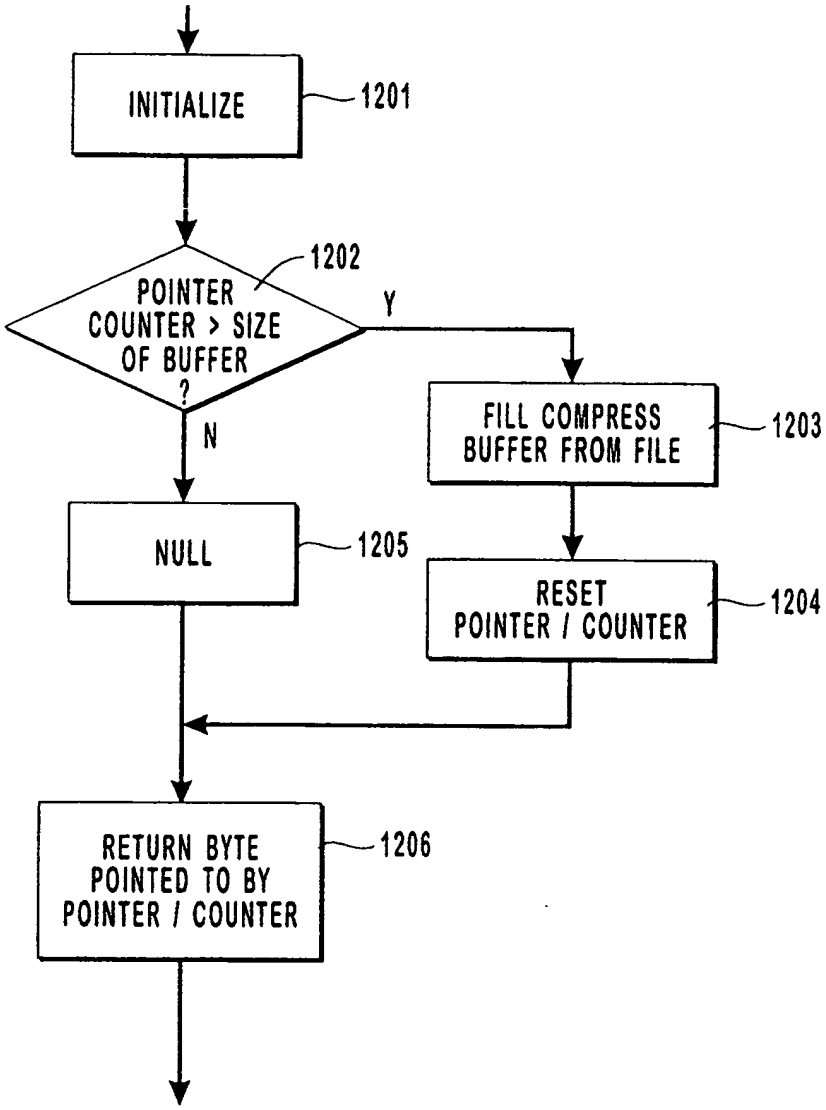

INITIALIZE
1201
POINTER COUNTER > SIZE OF BUFFER ?
1202
Y
N
FILL COMPRESS BUFFER FROM FILE
1203
NULL
1205
RESET POINTER / COUNTER
1204
RETURN BYTE POINTED TO BY POINTER / COUNTER
1206

FIG. 12

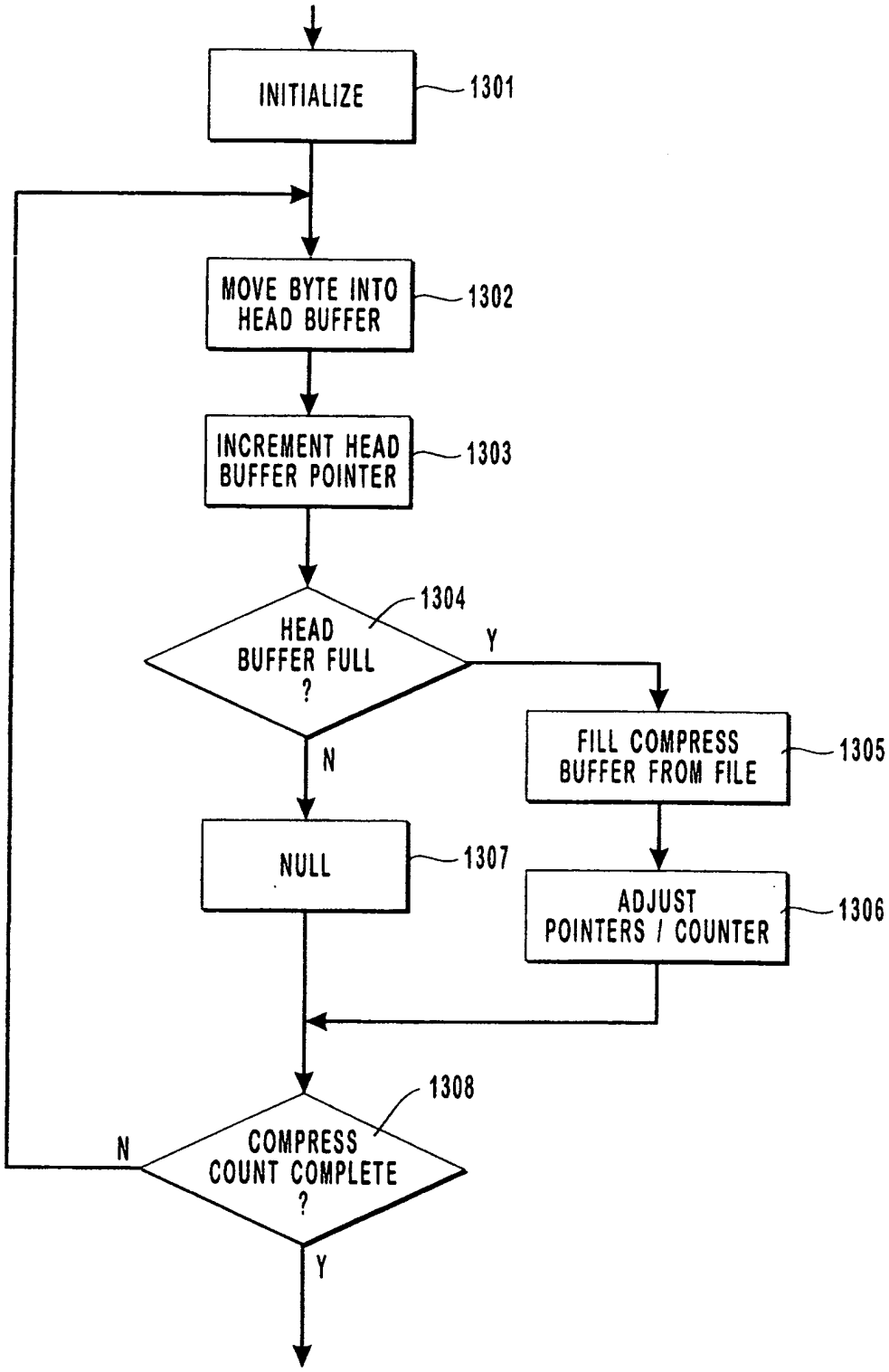

INITIALIZE
1301
MOVE BYTE INTO HEAD BUFFER
1302
INCREMENT HEAD BUFFER POINTER
1303
HEAD BUFFER FULL ?
1304
Y
N
FILL COMPRESS BUFFER FROM FILE
1305
NULL
1307
ADJUST POINTERS / COUNTER
1306
COMPRESS COUNT COMPLETE ?
1308
N
Y

FIG. 13

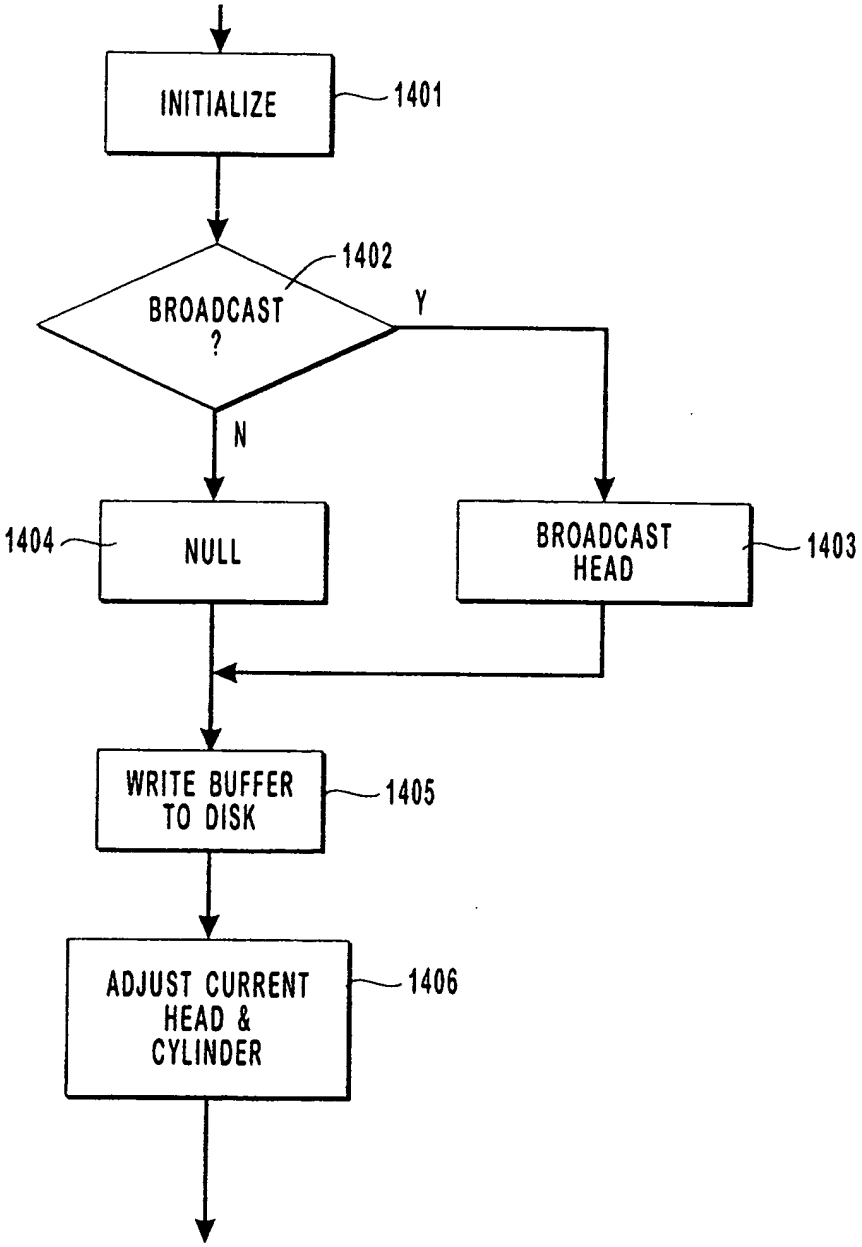

INITIALIZE
1401
BROADCAST ?
1402
Y
N
1404
NULL
BROADCAST HEAD
1403
WRITE BUFFER TO DISK
1405
ADJUST CURRENT HEAD & CYLINDER
1406

FIG. 14

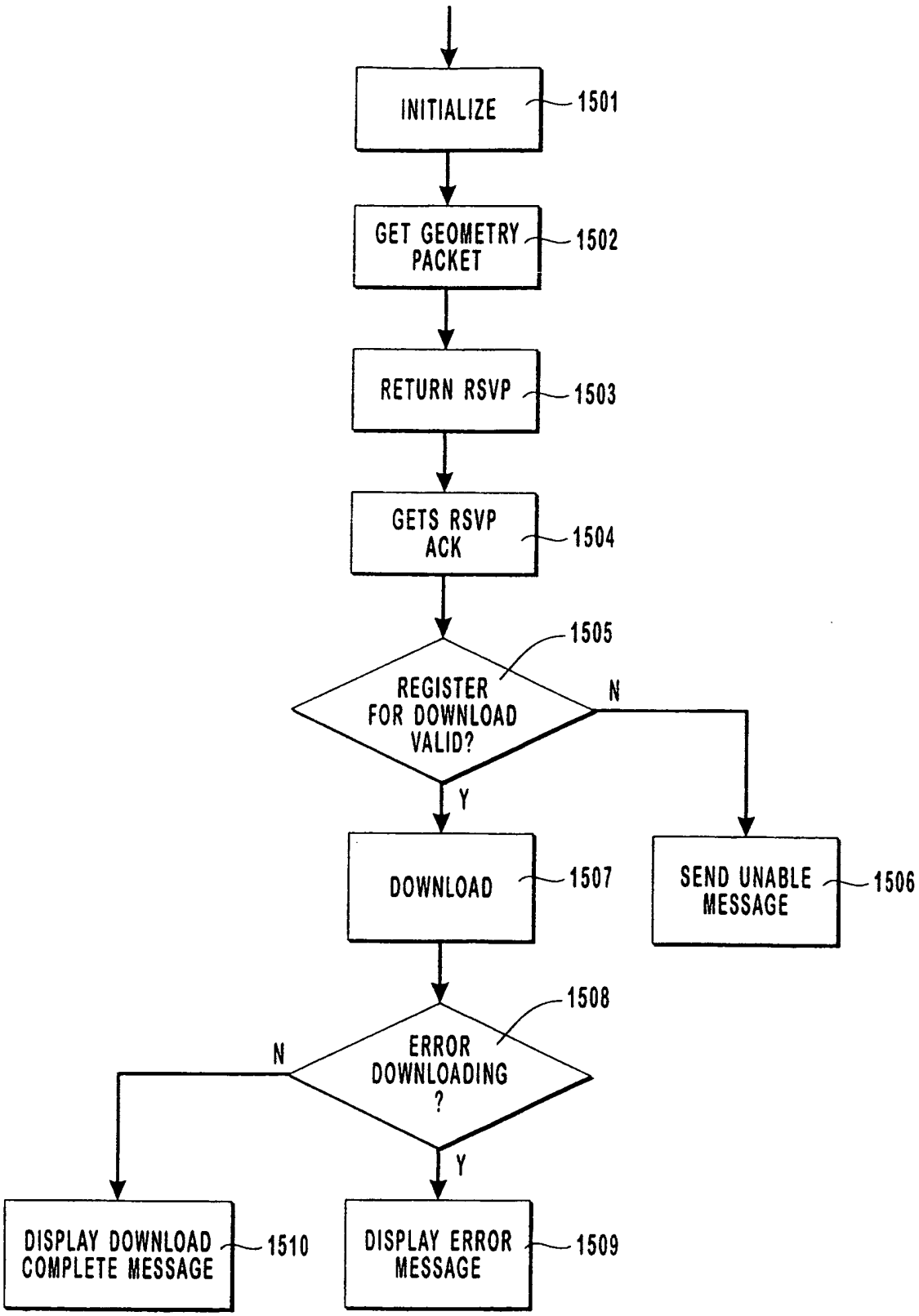

INITIALIZE
1501
GET GEOMETRY PACKET
1502
RETURN RSVP
1503
GETS RSVP ACK
1504
1505
REGISTER FOR DOWNLOAD VALID?
N
Y
DOWNLOAD
1507
SEND UNABLE MESSAGE
1506
1508
ERROR DOWNLOADING ?
N
Y
DISPLAY DOWNLOAD COMPLETE MESSAGE
1510
DISPLAY ERROR MESSAGE
1509

FIG. 15